conference

proceedings

3rd Symposium on Operating Systems Design and Implementation (OSDI '99) Proceedings

New Orleans, Louisiana
February 22–25, 1999

Sponsored by
The USENIX Association
Co-sponsored by **IEEE TCOS and ACM SIGOPS**

USENIX®

The Advanced Computing Systems Association

USENIX
Third Symposium on Operating Systems Design and Implementation (OSDI '99) Proceedings — *New Orleans, Louisiana February 1999*

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $23 for members and $30 for nonmembers.
Outside the U.S.A. and Canada, please add
$11 per copy for postage (via air printed matter).

Past OSDI Proceedings

OSDI '96 (Second)	October 1996	Seattle, Washington	$20/27
OSDI '94 (First)	November 1994	Monterey, California	$20/27

ISBN 1-880446-39-1

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)

Co-sponsored by IEEE TCOS and ACM SIGOPS

February 22-25, 1999
New Orleans, Louisiana

Program Committee

Program Co-Chairs

Margo Seltzer, *Harvard University*
Paul Leach, *Microsoft Corporation*

Program Committee

Tom Anderson, *University of Washington*
John Hartman, *University of Arizona*
Mike Jones, *Microsoft Corporation*
Kai Li, *Princeton University*
Bruce Lindsay, *IBM Almaden Research Center*
Nancy Lynch, *Massachusetts Institute of Technology*
Greg Minshall, *Siara Systems*
Sape Mullender, *University of Twente*
Michael O'Dell, *UUNET Technologies*
Sean O'Malley, *Network Appliance*
Rob Pike, *Lucent Technologies*

External Reviewers

Brian Bershad
Angelos Bilas
Peter Bosch
Miguel Castro
John Chapin
Fred Douglis
Dan Ellard
Yasuhiro Endo
Dawson Engler
Ed Felten
Jim Gray
Joseph M. Hellerstein
Eric Hoffman
Liviu Iftode
Frans Kaashoek
Dahlia Malkhi
Steve Manley
Clifford Neuman
Larry Peterson
David L. Presotto
David Redell
Stefan Savage
Harrick Vin
Jim Waldo
Dan Wallach
David Wetherall
David Wood
Alan Yoder

Contents

Third Symposium on Operating Systems Design and Implementation

February 22-25, 1999
New Orleans, Louisiana

Index of Authors vii
Message from the Program Chairs ix

Tuesday, February 23

I/O
Session Chair: Sean O'Malley, Network Appliance

Automatic I/O Hint Generation Through Speculative Execution 1
Fay Chang, Garth A. Gibson, Carnegie Mellon University

IO-Lite: A Unified I/O Buffering and Caching System 15
Vivek S. Pai, Peter Druschel, Willy Zwaenepoel, Rice University

Virtual Log Based File Systems for a Programmable Disk 29
Randolph Y. Wang, University of California, Berkeley; Thomas E. Anderson, University of Washington, Seattle; David A. Patterson, University of California, Berkeley

Resource Management
Session Chair: Greg Minshall, Siara Systems

Resource Containers: A New Facility for Resource Management in Server Systems 45
Gaurav Banga, Peter Druschel, Rice University; Jeffrey C. Mogul, Western Research Laboratory, Compaq Computer Corp.

Defending Against Denial of Service Attacks in Scout 59
Oliver Spatscheck, University of Arizona; Larry L. Peterson, Princeton University

Self-Paging in the Nemesis Operating System 73
Steven M. Hand, University of Cambridge Computer Laboratory

Wednesday, February 24

Kernels
Session Chair: Rob Pike, Lucent Technologies

Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System 87
Ben Gamsa, University of Toronto; Orran Krieger, IBM T.J. Watson Research Center; Jonathan Appavoo, Michael Stumm, University of Toronto

Interface and Execution Models in the Fluke Kernel 101
Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, Patrick Tullmann, University of Utah

Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels 117
Ariel Tamches, Barton P. Miller, University of Wisconsin, Madison

Real-Time
Session Chair: Mike Jones, Microsoft Corporation

ETI Resource Distributor: Guaranteed Resource Allocation and Scheduling in Multimedia Systems 131
Miche Baker-Harvey, Equator Technologies, Inc.

A Feedback-driven Proportion Allocator for Real-Rate Scheduling 145
David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, Jonathan Walpole, Oregon Graduate Institute

A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98 159
Erik Cota-Robles, James P. Held, Intel Architecture Labs

Distributed Systems
Session Chair: Tom Anderson, University of Washington

Practical Byzantine Fault Tolerance 173
Miguel Castro, Barbara Liskov, Massachusetts Institute of Technology

The Coign Automatic Distributed Partitioning System 187
Galen C. Hunt, Microsoft Research; Michael L. Scott, University of Rochester

Thursday, February 25

Virtual Memory
Session Chair: Kai Li, Princeton University

Tapeworm: High-Level Abstractions of Shared Accesses 201
Peter J. Keleher, University of Maryland

MultiView and Millipage—Fine-Grain Sharing in Page-Based DSMs 215
Ayal Itzkovitz, Assaf Schuster, Technion–Israel Institute of Technology

Optimizing the Idle Task and Other MMU Tricks 229
Cort Dougan, Paul Mackerras, Victor Yodaiken, New Mexico Institute of Technology

Filesystems
Session Chair: Bruce Lindsay, IBM Almaden Research Center

Logical vs. Physical File System Backup 239
Norman C. Hutchinson, University of British Columbia; Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, Sean O'Malley, Network Appliance, Inc.

The Design of a Multicast-based Distributed File System 251
Björn Grönvall, Assar Westerlund, Stephen Pink, Swedish Institute of Computer Science and Luleå University of Technology

Integrating Content-based Access Mechanisms with Hierarchical File Systems 265
Burra Gopal, Microsoft Corp; Udi Manber, University of Arizona

Index of Authors

Anderson, Thomas E. 29
Appavoo, Jonathan 87
Baker-Harvey, Miche 131
Banga, Gaurav 45
Castro, Miguel 173
Chang, Fay 1
Cota-Robles, Erik 159
Dougan, Cort 229
Druschel, Peter 15, 45
Federwisch, Michael 239
Ford, Bryan 101
Gamsa, Ben 87
Gibson, Garth A. 1
Goel, Ashvin 145
Gopal, Burra 265
Grönvall, Björn 251
Gruenberg, Joshua 145
Hand, Steven M. 73
Harris, Guy 239
Held, James P. 159
Hibler, Mike 101
Hitz, Dave 239
Hunt, Galen C. 187
Hutchinson, Norman C. 239
Itzkovitz, Ayal 215
Keleher, Peter J. 201
Kleiman, Steven 239
Krieger, Orran 87
Lepreau, Jay 101
Liskov, Barbara 173
Mackerras, Paul 229
Manber, Udi 265
Manley, Stephen 239
McGrath, Roland 101
McNamee, Dylan 145
Miller, Barton P. 117
Mogul, Jeffrey C. 45
O'Malley, Sean 239
Pai, Vivek S. 15
Patterson, David A. 29
Peterson, Larry L. 59
Pink, Stephen 251
Pu, Calton 145
Schuster, Assaf 215
Scott, Michael L. 187
Spatscheck, Oliver 59
Steere, David C. 145
Stumm, Michael 87
Tamches, Ariel 117
Tullmann, Patrick 101
Walpole, Jonathan 145
Wang, Randolph Y. 29
Westerlund, Assar 251
Yodaiken, Victor 229
Zwaenepoel, Willy 15

Message from the Program Chairs

We are happy to present the proceedings for the third Symposium on Operating System Design and Implementation, or OSDI III. To paraphrase a line of movie dialog: "Once is happenstance, twice is coincidence, the third time it's a tradition". With the third conference in a row having a program representing some of the best work being done in operating systems, we are declaring that a tradition has been created.

Once again, we coerced colleagues into serving on the program committee and were rewarded with an outstanding group -- 7 from industry and 6 from academia. They each read and reviewed an incredible number of papers and produced often extensive comments for the authors, whether or not the papers were accepted. We deeply appreciate all their efforts.

As with the second OSDI, we requested full papers instead of extended abstracts. These were reviewed in two rounds. In the first round, each paper was read by at least three members of the program committee, and often by external referees as well. At the end of the first round, about two-thirds of the papers were selected for review by at least another three program committee members.

Armed with reams of paper, the program committee met for a day and a half at Harvard University in Cambridge, Mass. By that time, papers that had high variance in the reviews had been read by even more committee members, and several papers that were heavily debated on the first day were read overnight by still more members. Many a committee member gave up valuable hours of sleep so that we'd have the very best data possible on all the submissions. By the end of the second day, we had selected 20 papers out of the 96 submitted.

Two of the accepted papers were co-authored by program committee members, out of six such submissions. As in the past, a significantly higher standard was applied to these submissions, and program committee authors did not know who their reviewers were, nor did they see their reviews or participate in discussions of their papers. In fact, they learned of the fate of their papers only a few hours before the rest of the authors did.

There was one paper selected for fast-track publication in the ACM Transactions on Computer Systems: "IO-Lite: A unified I/O buffering and caching system", by Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel, of Rice University. We thank Tom Anderson for serving as our official interface between OSDI and TOCS.

Another tradition, that of "shepherding" the accepted papers, was continued this year -- each paper was further revised after acceptance, based on the feedback from the reviews and with the guidance of a program committee member.

In addition to the refereed track, we have augmented the program with a Works-in-Progress session, organized by John Hartment, and an invited keynote by Jim Gettys. The keynote, "The Blind Men and The Elephant", discusses the intersection of the World Wide Web not only with operating systems but with other computing fields as well. We hope that these sessions will stimulate interesting and heated discussions both during and after the conference.

As is always the case, there are a number of people behind the scenes without whom such events never take place. Liz Pennell of Harvard University did an extraordinary amount of work processing the submissions, arranging for the program committee meeting, collating papers and reviews, interacting with authors, and picking up all the pieces at the end of the day. David Sullivan was the program committee scribe, taking notes of the committee's discussion so that authors would be provided as much feedback as possible. And, of course, there is the amazing Usenix staff: Judy DesHarnais, Ellie Young, Jane-Ellen Long, and Toni Veglia -- they make putting on a conference seem almost easy.

Last but certainly not least, we would like to heartily thank all the authors who submitted their work, without whom the tradition would not continue.

Paul J. Leach
Margo Seltzer

January, 1999

Automatic I/O Hint Generation through Speculative Execution

Fay Chang Garth A. Gibson

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{fwc,garth}@cs.cmu.edu

Abstract

Aggressive prefetching is an effective technique for reducing the execution times of disk-bound applications; that is, applications that manipulate data too large or too infrequently used to be found in file or disk caches. While automatic prefetching approaches based on static analysis or historical access patterns are effective for some workloads, they are not as effective as manually-driven (programmer-inserted) prefetching for applications with irregular or input-dependent access patterns. In this paper, we propose to exploit whatever processor cycles are left idle while an application is stalled on I/O by using these cycles to dynamically analyze the application and predict its future I/O accesses. Our approach is to speculatively pre-execute the application's code in order to discover and issue hints for its future read accesses. Coupled with an aggressive hint-driven prefetching system, this automatic approach could be applied to arbitrary applications, and should be particularly effective for those with irregular and, up to a point, input-dependent access patterns.

We have designed and implemented a binary modification tool, called "SpecHint", that transforms Digital UNIX application binaries to perform speculative execution and issue hints. TIP [Patterson95], an informed prefetching and caching manager, takes advantage of these application-generated hints to better use the file cache and I/O resources. We evaluate our design and implementation with three real-world, disk-bound applications from the TIP benchmark suite. While our techniques are currently unsophisticated, they perform surprisingly well. Without any manual modifications, we achieve 29%, 69% and 70% reductions in execution time when the data files are striped over four disks, improving performance by the same amount as manually-hinted prefetching for two of our three applications. We examine the performance of our design in a variety of configurations, explaining the circumstances under which it falls short of that achieved when applications were manually modified to issue hints. Through simulation, we also estimate how the performance of our design will be affected by the widening gap between processor and disk speeds.

This research is sponsored by DARPA/ITO through DARPA Order D306, and issued by Indian Head Division, NSWC under contract N00174-96-0002. Additional support was provided by an ONR graduate fellowship, and by the member companies of the Parallel Data Consortium, including: Hewlett-Packard Laboratories, Intel, Quantum, Seagate Technology, Storage Technology, Wind River Systems, 3Com Corporation, Compaq, Data General/Clariion, and Symbios Logic.

1 Introduction

Many applications, ranging from simple text search utilities to complex databases, issue large numbers of file access requests that cannot always be serviced by in-memory caches. Due to the disparity between processor speeds and disk access times, the execution times of these applications are often dominated by I/O latency. Furthermore, since disk access times are improving only slowly, these applications are receiving decreasing benefits from the rapid advance of processor technology, and I/O latency is accounting for an increasing proportion of their execution times.

File systems can automatically hide disk latency during file writes by performing write-behind buffering [Powell77], in which they inform the application that the write request has completed before propagating the data to disk. Automatically hiding the disk latency of file reads is more complicated since, in most applications, the requested data is used as soon as the read returns. *Prefetching*, requesting data before it is needed in order to move it from a high-latency locale (e.g. disk) to a low-latency locale (e.g. memory), is a well-known technique for hiding read latency. To be effective, prefetching requires that the I/O system provide more bandwidth than the application already consumes. Fortunately, we can construct cost-efficient I/O systems capable of providing adequate bandwidth by striping data across an array of disks [Patterson88] or, to facilitate sharing of I/O resources, across multiple higher-level entities like file servers or network disks [Cabrera91, Hartman94, Gibson98].

The difficulty with prefetching lies in knowing how to accurately determine what and when to prefetch. Prefetching consumes processor, cache and I/O resources; if unneeded data is prefetched, or data is prefetched prematurely, I/O requests for more immediately needed data may be delayed and/or more immediately needed data may be displaced from the file cache. One effective alternative is to manually modify applications so that they explicitly control I/O prefetching. Unfortunately, as we will discuss in the next section, this can be a difficult optimization problem for the program-

mer. Automatic prefetching, however, can significantly reduce execution time without increasing programming effort, provided that the automatic methods are sufficiently accurate, timely and careful with resource usage. In this paper, we present a novel approach to automatic prefetching that is potentially applicable to virtually all disk-bound applications and should be much more effective than existing automatic approaches for disk-bound applications with irregular and input-dependent access patterns.

Our approach arises from the observation that the cycles during which an application is stalled waiting for the I/O system to service a read request are often wasted. This situation occurs commonly both in desktop computing environments and where disk-bound applications are important enough to acquire exclusive use of a high-performance server machine. Even high-performance disk systems currently have at least 10 millisecond access latencies, so that processors may be wasting *millions* of cycles during each I/O stall. We propose that a wide range of disk-bound applications can use these cycles to dynamically discover their own future read accesses by performing *speculative execution*, a possibly erroneous pre-execution of their code.

We present a design for automatically transforming applications to perform speculative execution and issue hints for their future read accesses. Our design takes advantage of TIP [Patterson95], an informed prefetching and caching manager that uses application-generated hints to better exploit the file cache and I/O resources. We have implemented a binary modification tool, SpecHint, that performs this transformation. Using SpecHint, we obtain substantial reductions (29%, 69% and 70%) in the execution times of three real-world applications from the TIP benchmark suite [Patterson95] when the data is striped over four disks. For two of the three applications, we automatically obtain the same benefit as was obtained by manually modifying the applications to issue hints. We examine the performance of our design in a variety of configurations, explaining the circumstances under which it falls short of the performance achieved by manually-hinted prefetching. Through simulation, we also estimate how the performance of our design will be affected by the widening gap between processor and disk speeds.

This paper is organized as follows. In Section 2, we discuss previous prefetching mechanisms. In Section 3, we present our new automatic approach and our design for transforming applications. In Section 4, we describe our experimental framework and results. Finally, in Sections 5, 6, and 7, we present future work, related work, and conclusions.

2 Prefetching background

As mentioned in the introduction, applications can be manually modified to control I/O prefetching. For example, programmers can explicitly separate a request for data from the requirement that the data be available by issuing an asynchronous I/O call. However, there is a serious drawback to using asynchronous I/O. The size of the file cache, the latency and bandwidth of the I/O system, and the level of contention for the file cache and I/O system all affect the ideal scheduling of I/O requests. Issuing an asynchronous read call, however, causes the operating system to immediately issue a disk request for any uncached data specified by the call. Therefore, in redesigning an application to issue asynchronous I/O calls, a programmer implicitly makes assumptions about the characteristics of the systems on which the application will be executed.

Programmers can address this issue by using more sophisticated prefetching mechanisms, e.g. by modifying applications to issue hints for future read requests to a module that considers the dynamic I/O and caching behavior of the system before acting on the hint [Patterson94] (discussed further in Section 2.1). However, this does not avoid the higher-level problems with manual modification. First, manual modification requires that source code be available. Second, manual modification can involve formidable programming effort, both in understanding how the code currently generates read requests and in determining how the code should be modified so that the application will benefit from I/O prefetching. While some applications will only require the insertion of a few lines of code in a few strategic locations, other applications may require significant structural reorganization to support accurate and timely I/O prefetching [Patterson97]. Accordingly, we expect such modifications to be made only by a small fraction of programmers on a small fraction of programs. Therefore, automatic approaches are desirable.

The most widespread form of automatic I/O prefetching is the sequential read-ahead performed by most operating systems [Feiertag71, McKusick84] that exploits the preponderance of sequential whole-file reads [Ousterhout85, Baker91]. However, sequential read-ahead has limited utility when files are small. Furthermore, sequential read-ahead will not help, and may hurt, when access patterns are nonsequential.

In a more sophisticated history-based approach for automating I/O prefetching, the operating system gathers information about past file accesses and uses it to infer future file requests [Kotz91, Curewitz93, Griffioen94, Kroeger96, Lei97]. History-based prefetching is particularly well-suited for discovering and exploiting access patterns that span multiple applications. For example, it may implicitly recognize the edit-compile-run cycle

and prefetch the appropriate compiler, object files, or libraries while a user is editing a source file. When applied to disk-bound applications such as those used in our experiments, however, history-based approaches are less appropriate. These approaches are inherently limited by the tradeoff between the amount of history information retained and the achievable resolution in prefetching decisions. High resolution prediction – the ability to anticipate irregular block accesses in long-running disk-bound applications, for example – could require prohibitively large traces of prior executions. By whatever measures a particular history-based prefetching system reduces the amount of information it retains – e.g. by tracking only certain types of events or only the most frequently occurring events – the system will also sacrifice its ability to predict the accesses of applications whose access patterns vary widely between runs and/or applications that heavily exercise the I/O system but recur infrequently.

For these types of applications, we need a different approach for automating I/O prefetching. We would like an approach that considers precisely the factors which determine a specific application's stream of read requests, without burdening the operating system by requiring it to maintain long-term application-specific information. One such approach is for a tool, generally a compiler, to statically analyze an application in order to determine how read requests will be generated, and then transform the application so that the appropriate I/O prefetching will occur [Mowry96, Trivedi79, Cormen94, Thakur94, Paleczny95]. Such static approaches have proven extremely effective at reducing execution times for loop-intensive, array-based applications. However, these approaches are limited by hard interprocedural static analysis problems, especially because I/O is often an "outer loop" activity separated from the core computation by many layers of abstraction (procedure calls and jump tables, for example).

Our approach is based on having applications perform speculative execution, which is essentially a form of dynamic self-analysis. As with static approaches, we are able to capture application-specific factors which are expensive for history-based prefetching systems to extract and retain. Unlike static approaches, however, we do not require detailed understanding of the control and data flow of the application. Instead, our approach requires only a few simple static analyses and transformations. In addition, by relying on dynamic analysis, our approach can easily take advantage of input data values as they becomes available during the course of execution.

2.1 TIP

In the last section, we discussed why prefetching and caching decisions should depend on the dynamic state of the system. Patterson [Patterson94] and Cao [Cao94] have argued that this issue should be addressed by separating access understanding from resource allocation. Specifically, Patterson proposed that applications issue informing hints that disclose their future accesses as a sequence, allowing the underlying system to make optimal global decisions about what and when to prefetch, and what to eject from memory to make space for prefetched data. By issuing informing hints, applications would be both portable to other machines and sensitive to the changing conditions on any given machine.

Benchmark	Improvement	Description
`Agrep`	72%	text search
`Gnuld`	66%	object code linker
`XDataSlice`	70%	scientific visualization
`Davidson`	12%	computational physics
`Postgres, 20%`	48%	database join,
`Postgres, 80%`	69%	% tuples resulting
`Sphinx`	21%	speech recognition

Table 1: *Reductions in execution times using applications **manually modified** to issue hints for future accesses, as reported by Patterson [Patterson97]. These results were obtained on a 175MHz Digital 3000/600 with 128MB of memory running Digital UNIX 3.2c when the data was striped over four HP2247 disks with a 64KB striping unit.*

To validate his proposal, Patterson designed and built TIP, an informed prefetching and caching manager that replaces the Unified Buffer Cache manager in the Digital UNIX 3.2 kernel. TIP attempts to improve use of the file cache and I/O resources by performing a cost-benefit analysis. Roughly speaking, TIP estimates the benefit of prefetching in response to a hint based on the accuracy of previous hints from the application and the immediacy of the hint. It balances this estimated benefit against an estimated cost of prefetching, which is composed of the estimated cost of ejecting a block from the cache and the estimated opportunity cost of using the I/O system. On a benchmark suite that included a range of applications, informed prefetching and caching reduced execution times by 12-72% when data files were striped over four disks (see Table 1), clearly demonstrating that application-level hints for future read accesses can be effectively used to guide intelligent prefetching and caching decisions that take advantage of the bandwidth provided by a parallel I/O system.

These results are impressive, but the applications had to be manually modified to issue hints. For some of the applications, such as Gnuld and Sphinx, this involved significantly restructuring the code so that hints could be issued earlier and obtain more benefit from prefetching. The purpose of our research is to make the demonstrated benefits of prefetching readily accessible by automating the generation of informing hints.

Our design and implementation of speculative execution for automatic hint generation assumes that TIP is the underlying prefetching system (but could be retargetted to other prefetching systems). As shown in Table

Ioctl	Parameters	Description
TIPIO_SEG	batch of (filename, offset, length)	hints one or more segments from a named file
TIPIO_FD_SEG	batch of (file descriptor, offset, length)	hints one or more segments from an open file
TIPIO_CANCEL_ALL	none	cancels all outstanding hints from the issuing process

Table 2: *Relevant portion of the hinting interface exported by TIP. We do not exercise the capability for batching hints as speculative execution discovers reads one at a time. Recall that the standard UNIX read call takes a file descriptor, a pointer to a buffer, and a length as its parameters.*

2, TIP's hint interface includes calls which are almost directly analogous to the basic UNIX read calls. Our only modification of TIP was the addition of a CANCEL_ALL_HINTS call, which was accomplished with a few lines of code. The CANCEL_ALL_HINTS call will only cancel hints; once issued, prefetch requests cannot be cancelled.

3 Speculative execution

We propose that applications continue executing speculatively after they have issued a read request that misses in the file cache; that is, when they would ordinarily stall waiting for a disk read to complete. During this speculative execution, applications should issue the appropriate (non-blocking) hint call whenever they encounter a read request in order to inform the underlying prefetching system that the data specified by that request may soon be required. If the hinted data is not already cached and the prefetching system believes that prefetching the hinted data is the best use of disk and cache resources, then it should issue an I/O request for the hinted data. If the I/O system can parallelize fetching hinted data with its servicing of the outstanding read request, then the latency of fetching the data may be partially or completely hidden from the application.

Figure 1 depicts the intuition as to why speculative execution works. Consider an application which issues four read requests for uncached data and processes for a million cycles before each of these read requests. Assume that the data is distributed over three disks, that the disk access latency is three million cycles, and that there are sufficient cache resources to store all of the data used by this application once fetched. If we assume that speculative execution proceeds at the same pace as normal execution, then, while normal execution is stalled waiting for the first read request to complete, speculative execution may be able to issue hints for the remaining three read requests. If the data layout allows the hinted data to be fetched in parallel with service of the outstanding read request and the subsequent processing, then all of the subsequent read requests will hit in the cache, and the application's execution time will be more than halved.

Of course this is an oversimplification. Speculative execution will incur some run-time overhead. In addition, the pre-execution may be incorrect because some of the data values used during speculation may be incorrect (for example, those in the buffer into which data for

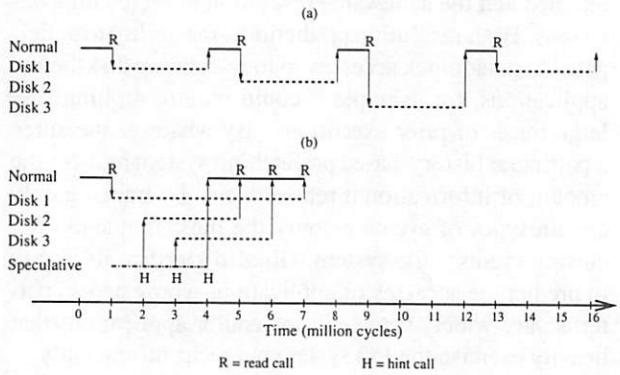


Figure 1: *Simplified example of how speculative execution reduces stall time: (a) shows how execution would normally proceed for a hypothetical application, and (b) shows how execution might proceed for the application if it performs speculative execution during I/O stalls in order to generate I/O hints. Performing speculative execution could more than halve the execution time of this example.*

the outstanding read request is being placed). Incorrect hints may lead the prefetching system to make erroneous prefetching and caching decisions. For example, they may result in the disks being busy reading unneeded data instead of servicing requests that are stalling the application, in keeping data in the cache that will not be needed but was identified by an incorrect hint, or in ejecting data from the cache that will be needed but was not identified by a hint. Furthermore, performing speculative execution will increase contention for other machine resources. This may result in normal (non-speculative) execution experiencing additional page faults, TLB misses and/or processor cache misses. Finally, if there is contention for the processor or the I/O system as, for example, with a multithreaded server or in a multiprogrammed environment, then speculative execution will have less opportunity to improve performance.

3.1 Design goals

We identify three basic design goals for how applications should be transformed to use speculative execution. Specifically, the transformation should be:

- *Correct* – the results of executing a transformed application should match those of executing the original application;
- *Free* – a transformed application should, at worst, be slower than the original application by an insignificant amount; and

- *Effective* – as many as possible of the application's requests for uncached data should be hinted in a timely fashion, with the minimum possible impact on machine resources.

3.2 Our design

Our design currently requires no specialized operating system support (other than the prefetching system and strictly prioritized kernel threads) and is appropriate for single-threaded applications. The basic element in our current design is the addition of a new kernel thread to the application. We call this thread the *speculating thread*, and its purpose is to perform speculative execution while the "original" application thread is stalled. We ensure that the speculating thread only executes when the original thread is stalled by assigning the speculating thread a low priority and selecting a preemptive scheduling policy which time-slices amongst only the highest priority runnable threads. A hint call is issued by the speculating thread whenever it encounters a read call.

3.2.1 Ensuring program correctness

There are three ways in which performing speculative execution could potentially change the behavior of the application. First, since the speculating thread shares an address space with the original thread, it could distort normal execution by changing code or data values that will be used by the original thread. Second, the speculating thread could produce side-effects visible outside the process, changing the impact of the application on the system. Finally, the speculating thread may inadvertently use inappropriate data values, like dividing by 0 or accessing an illegal address, that disrupt the execution of the application.

We ensure the correctness of our transformation by avoiding these potential problems. We prevent the speculating thread from producing side-effects visible outside the process by not allowing the speculating thread to issue any system calls except the hint calls (described in Table 2), and the `fstat()` and `sbrk()` calls.[1] We prevent the use of inappropriate data values from disturbing normal execution by installing signal handlers to catch any exceptions generated by the speculating thread, halting speculative execution until the original thread blocks on a new read call. Finally, we prevent the speculating thread from changing code or data values used by the original thread through *software-enforced copy-on-write*.

Inspired by software fault isolation [Wahbe93], software-enforced copy-on-write involves adding checks before each load and store instruction executed by the speculating thread, and adding a data structure to keep track of which memory regions have been copied and where their copies reside. Before each store instruction executed by the speculating thread, a check is added which accesses the data structure to discover whether the targetted memory region has already been copied. If so, the store is redirected to access the copy. If not, the memory region is copied, the data structure is updated, and the store is redirected to the newly created copy. Similarly, before each load instruction, a check is added which accesses the data structure to discover whether the referenced memory region has already been copied and, if so, redirects the load to obtain the value stored in the copy, which is the "current" value with respect to speculative execution.

Since load and store instructions comprise approximately 30% of the average instruction mix, software-enforced copy-on-write could be an expensive solution. For example, it may appear that the original thread would need to execute many additional branching instructions to avoid performing the checks. We avoid this overhead by making a complete copy of the binary's text section and constraining the speculating thread to only execute within the copy, which we call the *shadow code*. This permits us to add copy-on-write checks only around loads and stores in the shadow code, so that the original thread does not need to execute any additional instructions to support software-enforced copy-on-write.

Minimizing additional instructions in the original thread's code path is an example of our effort to minimize the *observable overhead* of supporting speculative execution. The checking necessary to perform software-enforced copy-on-write does not add directly to the execution time of the application; it simply causes speculative execution to proceed more slowly than normal execution; that is, it is *nonobservable overhead*. In general, we prefer design choices that incur nonobservable overhead to those that incur observable overhead since they seem less likely to affect worst-case performance.

We ensure that the speculating thread only executes shadow code by statically and/or dynamically checking and redirecting all control transfers (that is, possibilities for non-sequential changes in execution address). All control transfers that can be statically resolved are statically redirected to the appropriate address in the shadow code. Control transfers that cannot be statically resolved include those dynamically calculated using jump tables, corresponding to switch statements. Our binary modification tool only recognizes a few of the possible compiler-dependent jump table formats, so it can only statically handle switch statement control transfers that rely on jump tables in a recognized format. All other control transfers are statically redirected to call a special

[1] We add a set of memory allocation routines for use by the speculating thread to prevent speculative execution from introducing memory leaks. Notice that the behavior of an application could be inadvertently altered if it depends on its dynamic state (e.g. on the location of its sbrk pointer) or on the last access time of a file. We expect these types of applications to be uncommon.

handling routine with the originally intended target address as an argument. During runtime, if the originally intended target address is in the shadow code, the handling routine allows the speculating thread to proceed to that address. If the address is not in the shadow code but can be mapped to an address in the shadow code, then the handling routine redirects the speculating thread.[2] Otherwise, the handling routine simply prevents the speculating thread from leaving the shadow code (by preventing further progress until a new speculation is started, as discussed in the next section). Notice that, for applications with self-modifying code, this scheme will not allow the speculating thread to execute any newly created code, or to modify the existing shadow code.

One potential advantage of using software-enforced copy-on-write is the flexibility it permits in choosing the size of copy-on-write memory regions. However, when we explored this flexibility by varying the copy-on-write region size from 128B to 8192B, we discovered that it generally made no significant difference to the performance improvements obtained – the only difference larger than 5% was a 9% reduction in performance for Gnuld with a region size of 8192B. All of the results presented in this paper were obtained using 1024B regions.

3.2.2 Generating correct and timely hints

We would like to issue hints for as many of the read calls as possible so that TIP will have as much information as possible on which to base its prefetching and caching decisions. In addition, we would like to issue these hints as early as possible so that there will be ample opportunity to hide the latency of any prefetches. There are two situations that could obstruct these goals. First, because the speculating thread is only allowed to execute when the original thread is blocked, speculative execution could fall "behind" normal execution. If speculation is allowed to proceed in this situation, speculative execution would need to waste many cycles catching up to normal execution before it would be able to issue useful hints (that is, hints for read calls that have not already been issued). For some applications, including those with a long intermediate processing phase, speculative execution might never be able to catch up to normal execution. Second, because the speculating thread proceeds with incomplete state information, speculative execution could "stray" from the execution path that will be taken during normal execution. If speculation is allowed to proceed in this situation, the speculating thread might not be able to hint any future read calls. Even worse, it might generate a stream of incorrect hints, which could significantly hurt performance as explained at the beginning of Section 3. We describe speculative execution as being *on track* if the next hint issued would correctly predict the next unhinted future read call; otherwise, we describe speculative execution as being *off track*. We attempt to keep speculative execution on track as much as possible in order to increase the benefit we will be able to obtain through prefetching.

A pessimistic approach to keeping speculative execution on track would be to restart speculation every time the original thread blocks on a read call, where "restarting speculation" means causing the speculating thread to execute as if it had just returned from the call on which the original thread is currently blocked. However, this bounds how far speculative execution can predict the future to the distance it can progress during a single I/O stall, unnecessarily limiting the potential benefit of speculative execution. We attempt to increase the number of correct and timely hints generated by having the speculating and original threads cooperate to restart speculation only when they detect that speculative execution is off track.

Detecting when speculative execution is off track is accomplished by having the speculating thread record the hints it issues in a new data structure, called the *hint log*. The original thread maintains an index into the hint log and, whenever it is about to issue a read request, it checks the next entry in the hint log. If there is no next entry in the hint log, then the original thread knows that speculative execution is behind normal execution and is therefore off track. If there is an entry but it does not match the read request, then the original thread knows that speculative execution strayed from the correct execution path at some point in the past and is therefore off track. On the other hand, if the next entry matches the read, then, as far as the original thread can determine, speculative execution may still be on track.

Upon detecting that speculative execution is off track, the speculating and original threads also cooperate to restart speculation. In order to restart speculation, the speculating thread needs the original thread's state. When the original thread detects that speculative execution is off track, it copies the values of its registers into a data structure since the speculating thread cannot otherwise acquire their values and sets a "restart" flag to inform the speculating thread that it is off track. This work is performed before the original thread issues its read request because, if the original thread blocks on the read request, the speculating thread will have the opportunity to run. The speculating thread polls the restart flag frequently and, if the flag is set, cleans up its current speculation by cancelling any outstanding hints and clearing the copy-on-write data structure. The speculating thread then restarts speculation by loading the original thread's saved register values, making a copy of the

[2] Currently, the handling routine can only map function addresses, so that it can redirect control transfers through function pointers, but not computed goto statements.

original thread's stack[3], and jumping to the instruction which immediately follows the read system call in the shadow code.

Through this cooperation, we ensure that the speculating thread will not waste many cycles executing behind the original thread. We also ensure that the speculating thread will not waste cycles restarting speculative execution unless there is reason to believe that it is off track. While we cannot ensure that the speculating thread will not perform incorrect speculation and issue erroneous hints, we address this situation when it is detected by the original thread. Finally, we require the original thread to perform little additional work (at most, checking an entry in the hint log and saving its registers once per read) so that observable overhead is small.

3.3 Transforming applications

We use binary modification to automatically transform applications so that they will perform speculative execution. We chose to use binary modification because it does not require source code and can be both language- and compiler-independent. Of course, the speculative execution transformations could also be performed within a compiler.

The SpecHint tool is implemented in 16,000 lines of C code. Currently, the tool is relatively unsophisticated. It is restricted to Digital UNIX 3.2 Alpha binaries produced by the native `cc` compiler that are single-threaded, statically linked, and retain their relocation information. It does not yet perform any loop optimizations, which could significantly decrease the number of copy-on-write checks in some codes. The tool does recognize, and remove from the shadow code, calls to a few of the standard library output routines (`printf`, `fprintf` and `flsbuf`) because these routines are known not to influence future read accesses and can require many cycles to execute.

As illustrated in Figure 2, the application object files and libraries are first linked with the SpecHint auxiliary object files and the necessary libraries to support threading. The resulting binary is transformed by SpecHint, then linked normally to produces a transformed application executable. The SpecHint object files, which were generated from 4,000 lines of assembly code, include the dynamic memory allocation routines used by the speculating thread and the routine that handles control transfers that cannot be statically resolved (discussed in Section 3.2.1), as well as a routine that the speculating thread executes in order to restart speculation (discussed in Section 3.2.2). They also contain versions of `strncpy` and `memcpy` for the shadow code that were hand-optimized to simulate the effect of performing loop optimizations to minimize copy-on-write checks in these standard library routines.

[3] In combination with placing dynamic checks on instructions which modify the stack pointer and cannot be statically checked, copying the stack also allows us to avoid copy-on-write checks for load and store instructions off the stack pointer.

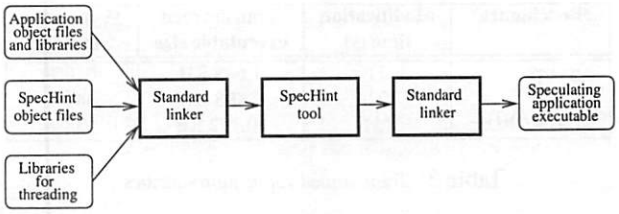


Figure 2: *Transforming applications to use speculative execution. The SpecHint object files contain various routines executed by the original or speculating thread in order to support speculative execution.*

4 Experimental evaluation

In this section, we describe our experimental environment, our benchmarks, and our results. The SpecHint tool implements the design described in the previous section by modifying Alpha binaries for Digital UNIX 3.2. Threading to support speculative execution was implemented using Digital UNIX's POSIX-compliant pthreads library.

Our experiments were conducted on an AlphaStation 255 (233MHz processor) with 256MB of main memory running Digital UNIX 3.2g, where the standard Unified Buffer Cache (UBC) manager was replaced with the TIP informed prefetching and caching manager. To facilitate comparison with Patterson's work [Patterson95], the file cache size was set to 12MB. The automatic read-ahead policy, which was invoked by all unhinted read calls, prefetches approximately the same number of blocks as have been sequentially read, up to a maximum of 64 blocks. The I/O system consisted of four HP C2247 disks (15ms average access times) attached by fast-wide-differential SCSI. Data files were striped over these disks by a striping pseudodevice with a striping unit of 64KB. A new file system was created to hold the files used in our experiments. All tests were run with the file cache initially empty. All reported results are averages over five runs. To facilitate comparison with programmer-inserted hints, we reran the manually modified applications [Patterson95] on this testbed.

4.1 Benchmark applications

We evaluated the effectiveness of our approach on three benchmark applications from the TIP benchmark suite [Patterson97].

Agrep (version 2.04) is a fast full-text pattern matching program. The application loops through the files specified on its command line, opening and reading each file sequentially. Therefore, the arguments to Agrep completely specify the stream of read accesses it will perform. In the benchmark, Agrep searches 1349 Digital

Benchmark	Modification time (s)	Transformed executable size	% increase in size
Agrep	21	1,648 KB	610%
Gnuld	23	2,408 KB	349%
XDataSlice	151	10,792 KB	138%

Table 3: *Transformed application statistics.*

UNIX kernel source files occupying 2928 disk blocks for a simple string that does not occur in any of the files.

Gnuld (version 2.5.2) is the Free Software Foundation's object code linker. The input object files are specified on the command line. Gnuld first reads each object file's file header, symbol header, symbol tables and string tables. The location of each file's symbol header is stored in its file header, and the locations of its symbol and string tables are stored in its symbol header. Gnuld then makes up to nine small, non-sequential reads in each object file to gather debugging information. The locations of these reads are determined from the symbol tables. Finally, Gnuld loops through the different non-debugging sections that appear in an object file, reading the corresponding section from each of the object files. Interspersed with the reads, Gnuld processes the data in order to produce and output an executable. In the benchmark, 562 binaries are linked to produce a Digital UNIX kernel.

XDataSlice (version 2.2) is a data visualization package that allows users to view a false-color representation of arbitrary slices through a three-dimensional data set. The original application limited itself to data sets that fit into memory, but Patterson modified the application to load data dynamically from large data sets [Patterson95]. In the benchmark, XDataSlice retrieves 25 random slices (the same slices used for Patterson's experiments) through a data set of 512^3 32-bit floating-point numbers that resides in 512MB of disk space.

4.2 Transformed applications

The application binaries were transformed by SpecHint on a 500MHz AlphaStation 500 with 1.5GB of memory. SpecHint is an unoptimized research prototype. Nevertheless, as shown in Table 3, SpecHint was able to modify our benchmark applications in a reasonable amount of time, 21 to 151 seconds. The resulting binaries were processed by the standard linker to produce speculating executables that, unlike the original application executables, contain shadow code, the SpecHint binaries, and libraries to support threading. These additions resulted in a 138% to 610% increase in executable size.

4.3 Overall performance results

As shown in Figure 3, performing speculative execution significantly reduces the execution times of our

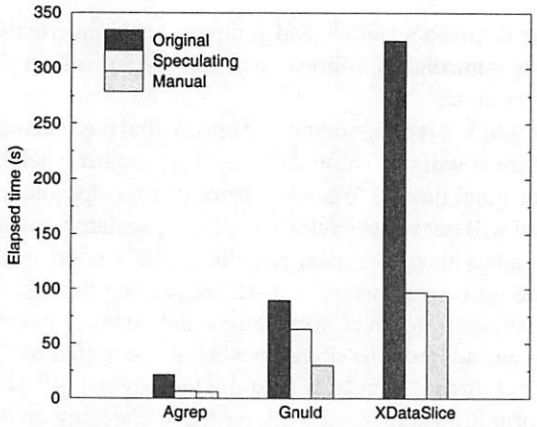


Figure 3: *Performance improvement.* Original *corresponds to the original, non-hinting applications;* Speculating *corresponds to the applications transformed to perform speculative execution for hint generation; and* Manual *corresponds to the applications manually modified to issue hints. In all cases, the non-hinted read calls issued by the applications invoked the operating system's sequential read-ahead policy (which is described at the beginning of Section 4).*

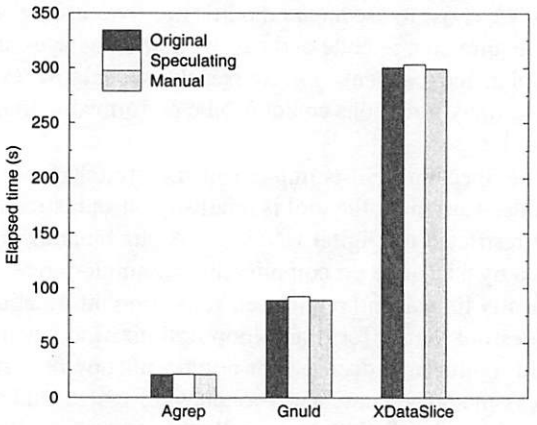


Figure 4: *Runtime overhead of supporting speculative execution, as captured by running the benchmarks with TIP configured to ignore hints.*

benchmark applications (by 69%, 29% and 70%, respectively, for Agrep, Gnuld and XDataSlice). For Agrep and XDataSlice, we were able to automatically achieve the same performance improvements obtained when the applications were manually modified to issue hints. For Gnuld, our gain was much less than that of the manually modified application, but still represents a substantial improvement over the original non-hinting application. Based on these results, and considering the relative unsophistication of our tool, speculative execution promises to be an effective technique for exploiting disk parallelism and underutilized processor cycles to reduce the execution time of disk-bound applications.

If TIP is configured to ignore hints, the applications that perform speculative execution were no more than 4%, and as little as 1%, slower than the original applications as shown in Figure 4. These figures capture all of the factors that can contribute to the worst-case perfor-

Benchmark		Read calls	Read blocks	Read bytes	Write calls	Write blocks	Write bytes
`Agrep`	total	4,277	2,928	18,091,527	0	0	0
	% hinted	68.1%	99.6%	99.7%	–	–	–
	inaccurately hinted	0	0	0	–	–	–
	% manually hinted	*68.3%*	*99.8%*	*>99.9%*	–	–	–
`Gnuld`	total	13,037	20,091	60,158,290	2343	3418	8,824,188
	% hinted	54.9%	67.5%	89.7%	–	–	–
	inaccurately hinted	2,336	6,721	37,177,440	–	–	–
	% manually hinted	*78.4%*	*86.0%*	*99.6%*	–	–	–
`XDataSlice`	total	46,356	46,352	370,663,914	2	2	4081
	% hinted	97.5%	97.5%	99.9%	–	–	–
	inaccurately hinted	0	0	0	–	–	–
	% manually hinted	*97.6%*	*97.6%*	*>99.9%*	–	–	–

Table 4: *Hinting statistics.* Total *includes explicit file calls only. The hinting behavior of the speculating applications is described by the* % hinted *and* inaccurately hinted *figures, and can be compared with the behavior of the manually modified applications (which issued no inaccurate hints) described by the* % manually hinted *figures. The number of read calls is sometimes larger than the number of read blocks because, for example, Agrep issues at least one extra read call per file to detect the end of the file. Discounting these non-data-returning reads (which do not need to be hinted), over 99% of Agrep's read calls were hinted.*

mance of the speculating applications except the potential negative effects of any erroneous hints. These factors include increased memory contention, the overhead of checking hint log entries before issuing read calls, and the overhead of executing an initialization routine that, among other things, spawns the speculating thread.

4.4 Performance analysis

Having established that speculative execution achieves significant performance improvements, we examine the behavior of the speculating applications and attempt to explain the differences between our results and those obtained with manually modified applications.

The primary metric for automatic hint generation is the number of correct hints generated. Table 4 summarizes the hinting behavior of the original and transformed applications. For Agrep and XDataSlice, we found that speculative execution was able to issue hints for nearly as many of the read calls as the manually modified applications. However, speculative execution was significantly less successful for Gnuld, hinting only 55% of the read calls in contrast to the 78% that the manually modified application was able to hint.

There are two basic reasons why speculating applications may hint fewer read calls than manually modified applications. One is that speculating applications must determine what to hint dynamically, but are only allowed to pursue hint discovery while normal execution is stalled. In fact, the more successfully a speculating application generates hints that will hide I/O latency, the less opportunity it will have to pursue hint discovery, unless the application is bandwidth-bound. The other reason is that data dependencies limit how early prefetches can be issued. For example, if the data specified by the next read call depends on the data returned by the currently outstanding read call, then speculative execution will not be able to hint the next read call.

Agrep is the most likely of our applications to be affected by the fact that hint discovery is only performed during I/O stalls. Agrep has the largest median number of cycles between read calls – 30362, 15902 and 4454 for Agrep, Gnuld and XDataSlice, respectively. It also has the largest ratio between the median number of cycles between hint calls and the median number of cycles between read calls – 7.5, 1.6 and 1.3 for Agrep, Gnuld and XDataSlice, respectively. (This ratio, which we call the *dilation factor*, is larger than one mainly due to the copy-on-write checks performed during speculative execution.) Accordingly, of our three applications, the speculating Agrep generates hints at by far the slowest rate. However, the almost equal gains achieved by the speculating Agrep and the manually modified Agrep indicate that this property of our design has negligible impact.

During the process of manually modifying an application to issue hints, programmers can make the application more amenable to prefetching by restructuring the code to increase the number of cycles between dependent read calls. As mentioned in Section 2.2, this was the case for the manually modified Gnuld. The speculating Gnuld, however, was produced from the original, unmodified code. It is only able to hint 55% of the read calls because a speculating application cannot hint a read call if it depends on a prior read and there are no I/O stalls between when the prior read completes and when the read call is issued. In addition, since a read cannot be hinted until all the data it is dependent on becomes available, data dependencies may cause hints to be issued too late to fully hide the latency of fetching the specified data. Comparing the speculating Gnuld to the manually modified Gnuld, over five times as many data blocks were only partially prefetched before being requested by the application (as shown in the *Partially* column of Table 5), indicating that the speculating Gnuld experienced many more I/O stalls. Finally, since each speculation proceeds with the assumption that future read calls are not data dependent, data dependencies may cause erroneous hints

Benchmark		Cache Block Reads	Prefetched Blocks	Prefetched Blocks						Cache Block Reuses
				Fully	%	Partially	%	Unused	%	
Agrep	Original	3,424	1,031	529	51.3%	499	48.4%	3	0.4%	416
	SpecHint	3,726	3,003	2,707	90.2%	272	9.1%	23	0.8%	655
	Manual	3,423	2,947	2,687	91.2%	258	8.8%	1	0.0%	421
Gnuld	Original	24,074	5,511	2,544	46.2%	2,014	36.6%	952	17.3%	12,435
	SpecHint	25,353	12,855	3,498	27.2%	5,432	42.3%	3,924	30.5%	13,646
	Manual	23,892	10,018	8,933	89.2%	1,057	10.6%	27	0.3%	13,519
XDataSlice	Original	49,997	60,702	12,806	21.1%	12,664	20.9%	35,231	58.0%	4,162
	SpecHint	50,810	45,338	40,319	88.9%	4,907	10.8%	112	0.3%	4,973
	Manual	49,782	44,938	40,167	89.4%	4,750	10.6%	20	0.0%	4,491

Table 5: *Prefetching and caching statistics. For the original, non-hinting applications, the prefetching figures are the result of the operating system's sequential read-ahead policy. For the speculating applications, the prefetching figures also include TIP's hint-driven prefetching.* Cache Block Reads *is the number of block reads from the file cache.* Prefetched Blocks *is the number of blocks prefetched from disk.* Fully *is the number of blocks whose prefetch completed before being requested by the application,* Partially *is the number of blocks partially prefetched before being requested by the application, and* Unused *is the number of prefetched blocks that were not accessed by the application before being ejected from the file cache. A* Cache Block Reuse *is counted each time a cached block services a second or subsequent request, and therefore indicates the effectiveness of caching. The closeness of the* Cache Block Reuse *figures indicates that erroneous prefetching did not significantly harm caching behavior.*

to be generated. The speculating Gnuld generates 2,336 erroneous hints, as shown in Table 4, contributing to the prefetching of 3,924 unused data blocks, as shown in Table 5.

Prefetching speculatively, and therefore sometimes incorrectly, is not new. History-based mechanisms all have this property. Specifically, Digital UNIX has an aggressive automatic read-ahead policy based on the expectation that files are read sequentially. It prefetches approximately the same number of blocks as have been read sequentially, up to a maximum of 64 blocks. For applications that issue nonsequential reads to large files, like XDataSlice, this read-ahead policy can be entirely too aggressive. As shown in Table 5, 58% of the blocks prefetched by sequential read-ahead for the non-hinting XDataSlice are not used. In contrast, since the read-ahead policy is only invoked by unhinted read calls and the hinting XDataSlices generate hints for almost all of the read calls, the hinting XDataSlices are able to almost eliminate the erroneous prefetches generated by the read-ahead policy.

4.5 Performance side-effects

In addition to generating hints, speculative execution will have other, less desirable performance effects. For example, since the speculating thread uses shadow code and performs copy-on-write, the speculating applications have larger memory footprints, consume memory more rapidly, and experience more page faults than the original applications. Table 6 shows that the memory footprints increase by 544 KB to 4.1 MB, the number of page reclaimes increases by 95 to 633, and the number of page faults increases by 12 to 40. In addition, the speculating applications may generate extraneous signals because speculative execution may use erroneous data in its calculations. Table 6 shows that the speculating applications generate up to 39 extraneous signals. However,

Benchmark		Footprint	Reclaims	Faults	Sigs
Agrep	Original	160 KB	39	4	0
	SpecHint	704 KB	134	16	0
	Manual	152 KB	39	4	0
Gnuld	Original	10.1 MB	1,341	12	0
	SpecHint	14.2 MB	1,974	52	39
	Manual	10.5 MB	1,389	14	0
XDS	Original	62.0 MB	8,105	61	0
	SpecHint	62.5 MB	8,202	93	2
	Manual	62.1 MB	8,104	60	0

Table 6: *Performance side-effects of speculative execution.* Footprint *is the maximum amount of memory that is physically mapped on behalf of the application at any time.* Reclaims *is the number of page reclaims, and* Faults *is the number of page faults, generated by the application. A page reclaim occurs if a referenced page is still in memory but is not physically mapped, and therefore requires operating system intervention but does not require a disk access. On our evaluation platform, at least one third of the memory-resident pages are not physically mapped, as determined by an LRU policy.* Sigs *is the number of signals generated by the application. For our applications, these signals were either segmentation violations or floating point exceptions.*

many of the additional page reclaims and page faults, and all of the additional signals, will occur while the original thread is blocked on I/O, so that they would be nonobservable overhead. As described in Section 4.3, the observable overhead of these performance side-effects is captured within the less than 4% increases in runtime observed when hints were disabled.

4.6 Varying file cache size

All previously reported results for the manually modified applications were obtained with a 12 MB file cache [Patterson95, Patterson97]. We measure the sensitivity of our results to the file cache size by running the benchmarks with a smaller (6 MB) file cache, and a larger (64MB) file cache. The cache size can affect performance because the sequential read-ahead policy sometimes prefetches data that will be accessed much later, and larger cache sizes may allow more of this data to remain in memory until the future access. For exam-

Benchmark		File cache size		
		6 MB	12 MB	64 MB
Agrep	Original	21.3	21.4	21.2
	SpecHint	6.5 (69%)	6.5 (70%)	6.4 (70%)
	Manual	6.3 (70%)	6.2 (71%)	6.1 (71%)
Gnuld	Original	106.3	89.5	56.5
	SpecHint	74.7 (30%)	63.3 (29%)	45.2 (20%)
	Manual	34.4 (68%)	30.2 (66%)	25.4 (55%)
XDS	Original	295.0	324.6	279.0
	SpecHint	94.6 (68%)	97.0 (70%)	87.8 (69%)
	Manual	91.4 (69%)	94.1 (71%)	85.8 (69%)

Table 7: *Elapsed time of applications as the file cache size is varied (in seconds). Percentages indicate performance improvement relative to the original, non-hinting application.*

ple, as shown in Table 7, the performance of the original, non-hinting Gnuld improves significantly as the cache size increases, reducing the benefit that can be obtained through prefetching. The speculating Gnuld achieves relatively less benefit with a 64MB cache because many of the read calls which it can generate hints for no longer require prefetching, whereas many of the read calls it is unable to hint continue causing I/O stalls. For Agrep and XDataSlice, there is little data reuse and sequential read-ahead seldom fetches data that is accessed much later, so the cache size does not affect the benefit obtained by the hinting applications.

4.7 Varying available I/O parallelism

While four-disk arrays are widely available, we also tested a single disk configuration and smaller and larger arrays. As shown in Table 8, the original, non-hinting applications are unable to derive much benefit from additional disks.

As shown in Figure 5, all the benchmarks receive significantly less benefit from speculative execution when there is only one disk because prefetching can only be overlapped with computation. The performance of speculating Gnuld degrades with one disk because erroneous prefetches consume scarce bandwidth, delaying service for the application's demand requests. As we discuss in Section 5, we believe that simple mechanisms can be employed to address this problem.

One objection to our assumptions – that disk-bound applications will be running on machines that have both disk arrays and no competing tasks to run on the processor – is that more than one disk is attached to a machine only if it is a shared server. However, Rochberg has shown that the TIP system can be effectively extended to allow clients to prefetch from distributed file servers with multiple disks [Rochberg97]. It is these "personal" clients that will be most rich in excess processor cycles.

As shown in Figure 5, the benefit of the hinting applications increase, and their runtimes decrease, when I/O parallelism is available. The benefit obtained by the manually modified applications increases monotonically

Benchmark	Number of Disks			
	1	2	4	10
Agrep	23.8	24.1	21.4	20.1
Gnuld	93.7	101.3	89.5	82.8
XDS	303.5	292.0	324.6	265.7

Table 8: *Elapsed time of original, non-hinting applications as the number of disks is varied (in seconds).*

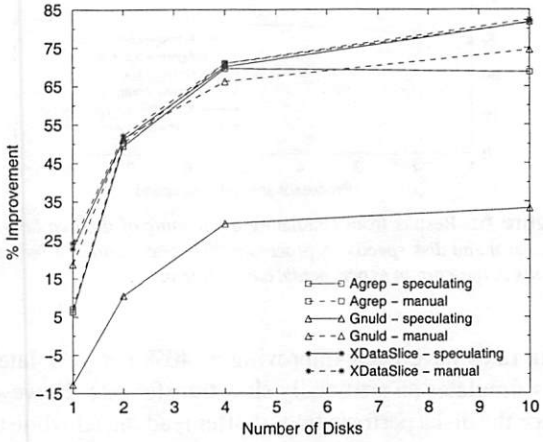


Figure 5: *Performance improvement as the number of disks is varied.*

with the number of disks since these applications always issue enough hints to take advantage of the additional disks. For Agrep, the benefit of the speculating application mirrors that of the manually modified application for the 2 and 4 disk configurations. However, due to the dilation factor discussed in Section 4.4, speculative execution is not far enough ahead of normal execution to issue sufficient hints to keep 10 disks busy. For Gnuld, data dependencies limit hint generation, and therefore the degree to which the speculating application is able to utilize additional disks. For XDataSlice, however, speculative execution generates more than enough hints to take advantage of the additional I/O parallelism.

4.8 Increasing relative processor speed

Due to rapid improvements in processor technology, the gap between processor speeds and I/O latency continues to widen. This will increase the number of cycles per I/O stall, and therefore the progress that speculative execution can make during a single stall. To predict the impact of this trend on the effectiveness of our approach, we modified the striping pseudodevice to delay notification of completed I/O requests. For example, to simulate the effect of doubling the gap between processor and disk speeds, we doubled the time before the system was notified that each I/O request had completed, then scaled our resulting measurements by half.[4] Since disk positioning times and data rates improve at different rates, and

[4]To obtain the desired effect on the perceived service time of prefetch requests, we configured the pseudodevice to limit the number of prefetch requests outstanding at each disk to at most one.

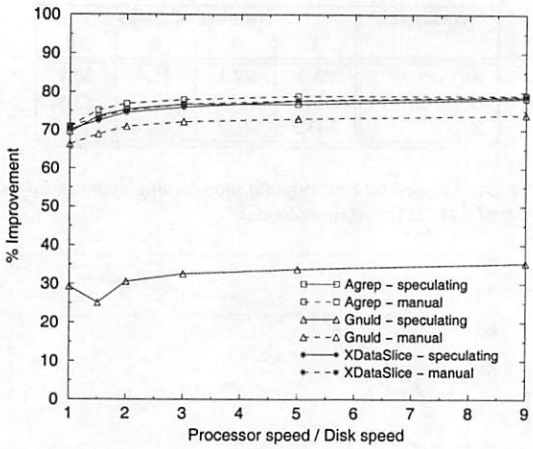


Figure 6: *Results from simulating a widening of the gap between processor and disk speeds. A processor/disk speed ratio of 1 indicates results in our current experimental environment.*

data rates have been improving at 40% per year lately, this simulates an artificially slow transfer rate. However, since the disks perform track-buffer read-ahead while the pseudodevice is delaying completion, accesses which are physically sequential will appear to have a faster than modelled transfer rate.

Our simulation results are shown in Figure 6. The improvements obtained by the manually modified applications increase steadily but insignificantly. This is unsurprising since their performance is limited by the available I/O bandwidth and their processing times are already only a small percentage of their execution times. The curves for the speculating applications are similar to those for the manually modified applications, although offset in Gnuld's case. For Agrep and XDataSlice, speculative execution already generates enough hints to keep the disks busy at all times.[5] For Gnuld, data dependencies, which are independent of processor speed, prevent speculative execution from using the additional cycles during I/O stalls to hint more read calls. For some applications, a more sophisticated design may be able to take advantage of these additional cycles. For example, it may prove useful to loosen our current definition of what it means for speculative execution to be on track. In general, however, applications dependent on recently read values may not be able to derive additional benefit from faster processors (unless they are rewritten to allow newly read data to affect future reads only after more intervening disk requests have been issued).

[5]Recall from the last section that the speculating Agrep was not able to generate enough hints to keep 10 disks busy on our current experimental platform. Under simulation, increasing the processor-to-disk speed ratio alleviated this problem so that, with a ratio of 3, the performance improvement of the speculating Agrep and the manually modified Agrep were 87% and 84%, respectively.

5 Future work

Having successfully demonstrated that speculative execution can be used to automate I/O hint generation, we are working on refining our design to better handle data-dependent applications like Gnuld. We discovered that even a simple, ad-hoc mechanism – disabling speculative execution for a brief time after some number of cancel requests have been issued – was sufficient to eliminate the performance penalty of performing speculative execution in Gnuld when the I/O system offered no parallelism. We are exploring more generic methods for limiting the number of erroneous hints generated, and for reducing the negative impact of erroneous hinting.

We are also investigating how speculative execution can be effectively employed in the range of possible multiprogramming/multithreaded scenarios. In particular, we are developing methods for evaluating the effectiveness of any particular speculation and for using this evaluation to decide what speculation, if any, should be scheduled and allowed to consume shared machine resources.

Multiprocessor environments offer another exciting possibility. One of the biggest challenges for proponents of multiprocessors is how they will enable non-parallelized applications to utilize the additional processing resources. By performing speculative execution in parallel with normal execution, disk-bound applications that cannot be automatically parallelized using compiler techniques may still be able to take advantage of the additional processing capabilities of a multiprocessor.

6 Related work

In Section 2, we discussed history-based prefetching, static approaches to automating prefetching, informing hints and the TIP prefetching and caching manager.

Mowry, Demke and Krieger's work [Mowry96] relies on static analysis, but also makes use of dynamic information provided by the operating system. Their approach applies to memory-mapped files, so that their hints affect virtual memory management as well as file cache management. Their use of hints differs from ours in that their compiler is responsible for placing hints based on a static decision of when prefetches should be issued, whereas we rely on TIP to manage the scheduling of prefetches.

Research presented by Franaszek, Robinson and Thomasian is close in spirit to our own [Franaszek92]. Through simulation, they demonstrated that pre-executing database transactions in order to prefetch data or pre-claim locks could significantly increase throughput because it reduced effective concurrency. However, their simulations assumed that pre-execution would always cause the correct data to be prefetched (or the correct locks to be claimed). Our approach differs from

theirs primarily in two aspects. First, to reduce conflicts, they proposed that pre-execution of a transaction would run to completion before the transaction would re-execute with the intent to commit. In our system, pre-execution is overlapped with, and always secondary to, normal execution. Second, they explored pre-execution as a concurrency control technique for manual inclusion in the design and implementation of database systems. One of the essential properties of our work is the ability to automatically transform applications to use pre-execution.

The idea of adding software checks around load and store instructions was first brought to our attention by Lucco and Wahbe [Wahbe93]. They used these checks to perform software fault isolation, a fast alternative to hardware-enforced memory protection. Our checks are more complex and costly in order to implement software-enforced copy-on-write.

7 Conclusions

Disk-bound applications, increasingly common as faster computers and larger storage encourage users to manipulate more data, have their performance determined by storage rather than processor performance. While parallel storage systems are increasingly common, applications that exploit them well are not. Aggressive prefetching is a simple way to effectively utilize storage parallelism to reduce application latency, provided sufficiently detailed predictions of future accesses can be made sufficiently early.

This paper extends aggressive prefetching research with an automatic hint generation technique based on speculative pre-execution using mid-execution application state. Invoked only when the application is stalled waiting for I/O, speculative execution can add little or no observable overhead to the application. Provided that cycles are available in these time periods, speculative execution can discover future read accesses and issue hints to an aggressive prefetching system.

We have designed and implemented a binary modification tool that transforms Digital UNIX binaries to automatically perform speculative execution. Applied to a text search utility, a linker, and a 3-D visualization program, our system demonstrated 29% to 70% reductions in execution time with a four-disk array. A principle limitation of the current design is the lack of more effective automatic mechanisms for limiting the penalty of erroneous hinting due to data dependencies. The relatively large success of our currently unsophisticated design demonstrates that speculative execution is a promising new approach to aggressive I/O prefetching.

Acknowledgements

We thank David Nagle and Digital Equipment Corporation for providing the AlphaStation 500. We thank Paul Mazaitis for setting up the various hardware configurations, David Rochberg and Jim Zelenka for their assistance with TIP and Digital UNIX, and Robert O'Callahan for many invaluable discussions. We also thank John Hartman and the anonymous referees for their feedback on earlier drafts of this paper. TIP was developed by Hugo Patterson, and the SpecHint tool was inspired by a project with Steve Lucco to implement a software fault isolation tool for Digital UNIX.

References

[Baker91] Mary Baker, et. al. Measurements of a distributed file system. Proceedings of the 13th SOSP, October 1991.

[Cabrera91] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. Computing Systems 4(4), pp.405-436, Fall 1991.

[Cao94] P. Cao, E.W. Felton and K. Li. Implementation and performance of application-controlled file caching. Proceedings of the 1st OSDI. November, 1994.

[Cormen94] ViC*: A preprocessor for virtual-memory C*. TR PCS-TR94-243, Department of Computer Science, Dartmouth College, November 1994.

[Curewitz93] K.M. Curewitz, P. Krishnan and J.S. Vitter. Practical prefetching via data compression. Proceedings of the 1993 SIGMOD, May 1993.

[Feiertag71] The Multics input/output system. Proceedings of the 3rd SOSP, 1971.

[Franaszek92] P.A. Franaszek, J.T. Robinson and A. Thomasian. Concurrency control for high contention environments. ACM TODS, V 17(2), pp. 304-345, June 1992.

[Gibson98] Garth Gibson, et. al. A cost-effective, high-bandwidth storage architecture. Proceedings of the 8th ASPLOS. October, 1998.

[Griffioen94] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. Proceedings of 1994 Summer USENIX, June 1994.

[Hartman94] John H. Hartman. The Zebra striped network file system. Doctoral thesis, UCB/CSD-95-867, December 1994.

[Kotz91] David Kotz and Carla Ellis. Practical prefetching techniques for parallel file systems. Proceedings of the 1st PDIS, December 1991.

[Kroeger96] T. Kroeger and D. Long. Predicting file system actions from prior events. Proceedings of 1996 Winter USENIX, January 1996.

[Lei97] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. Proceedings of the 1996 Winter USENIX, January 1997.

[McKusick84] M.K. McKusick, et. al. A fast file system for UNIX. ACM TOCS, V 2(3), pp. 181-197, August 1984.

[Mowry96] Todd Mowry, Angela Demke and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. Proceedings of the 2nd OSDI, October 1996.

[Ousterhout85] J.K. Ousterhout, et. al. A trace-driven analysis of the UNIX 4.2 BSD file system. Proceedings of the 10th SOSP, December 1985.

[Paleczny95] M. Paleczny, K. Kennedy and C. Koelbel. Compiler support for out-of-core arrays on data parallel machines. Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation, February 1995.

[Patterson88] David Patterson, Garth Gibson and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). Proceedings of the 1988 SIGMOD. June 1988.

[Patterson94] Hugo Patterson and Garth Gibson. Exposing I/O concurrency with informed prefetching. Proceedings of the 3rd PDIS. September, 1994.

[Patterson95] Hugo Patterson, et. al. Informed prefetching and caching. Proceedings of the 15th SOSP. December, 1995.

[Patterson97] Hugo Patterson. Informed prefetching and caching. Doctoral Thesis, CMU-CS-97-204, December 1997.

[Powell77] Michael L. Powell. The DEMOS file system. Proceedings of the 6th SOSP, November 1977.

[Rochberg97] David Rochberg and Garth Gibson. Prefetching over a network: Early experience with CTIP. ACM SIGMETRICS Performance Evaluation Review, V 25(3), pp. 29-36, December 1997.

[Thakur94] R. Thakur, R. Bordawekar and A. Choudhary. Compilation of out-of-core data parallel programs for distributed memory machines. Workshop on I/O in Parallel Computer Systems, IPPS94, April 1994.

[Trivedi79] K.S. Trivedi. An analysis of prepaging. Computing, V 22(3), pp.191-210, 1979.

[Wahbe93] Robert Wahbe, et. al. Efficient software-based fault isolation. Proceedings of the 14th SOSP, December 1993.

IO-Lite: A Unified I/O Buffering and Caching System

Vivek S. Pai‡ Peter Druschel† Willy Zwaenepoel†
‡ *Department of Electrical and Computer Engineering*
† *Department of Computer Science*
Rice University

Abstract

This paper presents the design, implementation, and evaluation of IO-Lite, a unified I/O buffering and caching system for general-purpose operating systems. IO-Lite unifies *all* buffering and caching in the system, to the extent permitted by the hardware. In particular, it allows applications, interprocess communication, the filesystem, the file cache, and the network subsystem to share a single physical copy of the data safely and concurrently. Protection and security are maintained through a combination of access control and read-only sharing. IO-Lite eliminates all copying and multiple buffering of I/O data, and enables various cross-subsystem optimizations. Experiments with a Web server on IO-Lite show performance improvements between 40 and 80% on real workloads.

1 Introduction

For many users, the perceived speed of computing is increasingly dependent on the performance of networked server systems, underscoring the need for high performance servers. Unfortunately, general purpose operating systems provide inadequate support for server applications, leading to poor server performance and increased hardware cost of server systems.

One source of the problem is lack of integration among the various input-output (I/O) subsystems and the application in general-purpose operating systems. Each I/O subsystem uses its own buffering or caching mechanism, and applications generally maintain their own private I/O buffers. This approach leads to repeated data copying, multiple buffering of I/O data, and other performance-degrading anomalies.

Repeated data copying causes high CPU overhead and limits the throughput of a server. Multiple buffering of data wastes memory, reducing the space available for the document cache. This size reduction causes higher cache miss rates, increasing disk accesses and reducing throughput. Finally, lack of support for application-specific cache replacement policies [9] and optimizations like TCP checksum caching [15] further reduce server performance.

We present the design, the implementation, and the performance of IO-Lite, a unified I/O buffering and caching system for general-purpose operating systems. IO-Lite unifies *all* buffering and caching in the system to the extent permitted by the hardware. In particular, it allows applications, interprocess communication, the file cache, the network subsystem, and other I/O subsystems to share a single physical copy of the data safely and concurrently. IO-Lite achieves this goal by storing buffered I/O data in immutable buffers, whose locations in physical memory never change. The various subsystems use mutable buffer aggregates to access the data according to their needs.

The primary goal of IO-Lite is to improve the performance of server applications such as those running on networked (e.g., Web) servers, and other I/O-intensive applications. IO-Lite avoids redundant data copying (decreasing I/O overhead), avoids multiple buffering (increasing effective file cache size), and permits performance optimizations across subsystems (e.g., application-specific file cache replacement and cached Internet checksums).

A prototype of IO-Lite was implemented in FreeBSD. In keeping with the goal of improving performance of networked servers, our central performance results involve a Web server, in addition to other benchmark applications. Results show that IO-Lite yields a performance advantage of 40 to 80% on real workloads. IO-Lite also allows efficient support for dynamic content using third-party CGI programs without loss of fault isolation and protection.

1.1 Background

In state-of-the-art, general-purpose operating systems, each major I/O subsystem employs its own buffering and caching mechanism. In UNIX, for instance, the network subsystem operates on data stored in BSD *mbufs* or the equivalent System V *streambufs*, allocated from a private kernel mem-

ory pool. The mbuf (or streambuf) abstraction is designed to efficiently support common network protocol operations such as packet fragmentation/reassembly and header manipulation.

The UNIX filesystem employs a separate mechanism designed to allow the buffering and caching of logical disk blocks (and more generally, data from block oriented devices.) Buffers in this *buffer cache* are allocated from a separate pool of kernel memory.

In older UNIX systems, the buffer cache is used to store all disk data. In modern UNIX systems, only filesystem metadata is stored in the buffer cache; file data is cached in VM pages, allowing the file cache to compete with other virtual memory segments for the entire pool of physical main memory.

No support is provided in UNIX systems for buffering and caching at the user level. Applications are expected to provide their own buffering and/or caching mechanisms, and I/O data is generally copied between OS and application buffers during I/O read and write operations[1]. The presence of separate buffering/caching mechanisms in the application and in the major I/O subsystems poses a number of problems for I/O performance:

Redundant data copying: Data copying may occur multiple times along the I/O data path. We call such copying *redundant*, because it is not necessary to satisfy some hardware constraint. Instead, it is imposed by the system's software structure and its interfaces. Data copying is an expensive operation, because it generally proceeds at memory rather than CPU speed and it tends to pollute the data cache.

Multiple buffering: The lack of integration in the buffering/caching mechanisms may require that multiple copies of a data object be stored in main memory. In a Web server, for example, a data file may be stored in the filesystem cache, in the Web server's buffers, and in the network subsystem's send buffers of one or more connections. This duplication reduces the effective size of main memory, and thus the size and hit rate of the server's file cache.

Lack of cross-subsystem optimization: Separate buffering mechanisms make it difficult for individual subsystems to recognize opportunities for optimizations. For example, the network subsystem of a server is forced to recompute the Internet checksum each time a file is being served from the server's cache, because it cannot determine that the same data is being transmitted repeatedly. Also, server applications cannot exercise customized file cache replacement policies.

The outline of the rest of the paper is as follows: Section 2 presents the design of IO-Lite and discusses its operation in a Web server application. Section 3 describes a prototype implementation in a BSD UNIX system. A quantitative evaluation of IO-Lite is presented in Section 4, including performance results with a Web server on real workloads. In Section 5, we present a qualitative discussion of IO-Lite in the context of related work, and we conclude in Section 6.

2 IO-Lite Design

2.1 Principles: Immutable Buffers and Buffer Aggregates

In IO-Lite, all I/O data buffers are *immutable*. Immutable buffers are allocated with an initial data content that may not be subsequently modified. This access model implies that all sharing of buffers is read-only, which eliminates problems of synchronization, protection, consistency, and fault isolation among OS subsystems and applications. Data privacy is ensured through conventional page-based access control.

Moreover, read-only sharing enables very efficient mechanisms for the transfer of I/O data across protection domain boundaries, as discussed in Section 2.2. For example, the filesystem cache, applications that access a given file, and the network subsystem can all safely refer to a single physical copy of the data.

The price for using immutable buffers is that I/O data can not generally be modified in place[2]. To alleviate the impact of this restriction, IO-Lite encapsulates I/O data buffers inside the *buffer aggregate* abstraction. Buffer aggregates are instances of an abstract data type (ADT) that represent I/O data. All OS subsystems access I/O data through this unified abstraction. Applications that wish to obtain the best possible performance can also choose to access I/O data in this way.

The data contained in a buffer aggregate does not generally reside in contiguous storage. Instead, a buffer aggregate is represented internally as an ordered list of <*pointer,length*> pairs, where each pair refers to a contiguous section of an immutable I/O buffer. Buffer aggregates support operations for truncating, prepending, appending, concatenating, splitting, and mutating data contained in I/O buffers.

[1] Some systems partly avoid this data copying under certain conditions in a transparent manner, using page remapping and copy-on-write.

[2] As an optimization, I/O data can be modified in place if it is not currently shared.

While the underlying I/O buffers are *immutable*, buffer aggregates are *mutable*. To mutate a buffer aggregate, modified values are stored in a newly allocated buffer, and the modified sections are then logically joined with the unmodified portions through pointer manipulations in the obvious way. The impact of the absence of in-place modifications will be discussed in Section 2.8.

In IO-Lite, all I/O data is encapsulated in buffer aggregates. Aggregates are passed among OS subsystems and applications by value, but the associated IO-Lite buffers are passed *by reference*. This approach allows a single physical copy of I/O data to be shared throughout the system. When a buffer aggregate is passed across a protection domain boundary, the VM pages occupied by all of the aggregate's buffers are made readable in the receiving domain.

Conventional access control ensures that a process can only access I/O buffers associated with buffer aggregates that were explicitly passed to that process. The read-only sharing of immutable buffers ensures fault isolation, protection, and consistency despite the concurrent sharing of I/O data among multiple OS subsystems and applications. A system-wide reference counting mechanism for I/O buffers allows safe reclamation of unused buffers.

2.2 Interprocess Communication

In order to support caching as part of a unified buffer system, an interprocess communication mechanism must allow safe *concurrent* sharing of buffers. In other words, different protection domains must be allowed protected, concurrent access to the same buffer. For instance, a caching Web server must retain access to a cached document after it passes the document to the network subsystem or to a local client.

IO-Lite uses an IPC mechanism similar to *fbufs* [11] to support safe concurrent sharing. Copy-free I/O facilities that only allow *sequential* sharing [21, 7] are not suitable for use in caching I/O systems, since only one protection domain has access to a given buffer at any time and reads are destructive.

I/O-Lite extends fbufs in two significant directions. First, it extends the fbuf approach from the network subsystem to the filesystem, including the file data cache, thus unifying the buffering of I/O data throughout the system. Second, it adapts the fbuf approach, originally designed for the x-kernel [13], to a general-purpose operating system.

IO-Lite's IPC, like fbufs, combines page remapping and shared memory. Initially, when an (immutable) buffer is transferred, VM mappings are updated to grant the receiving process read access to the buffer's pages. Once the buffer is deallocated, these mappings persist, and the buffer is added to a cached pool of free buffers associated with the I/O stream on which it was first used, forming a lazily established pool of read-only shared memory pages.

When the buffer is reused, no further VM map changes are required, except that temporary write permissions must be granted to the producer of the data, to allow it to fill the buffer. This toggling of write permissions can be avoided whenever the producer is a trusted entity, such as the OS kernel. Here, write permissions can be granted permanently, since a trusted entity can be implicitly expected to honor the buffer's immutability.

IO-Lite's worst case cross-domain transfer overhead is that of page remapping; it occurs when the producer allocates the last buffer before the first buffer is deallocated by the receiver(s). Otherwise, buffers can be recycled, and the transfer performance approaches that of shared memory.

2.3 Access Control and Allocation

IO-Lite ensures access control and protection at the granularity of processes. That is, no loss of security or safety is associated with the use of IO-Lite. IO-Lite maintains cached pools of buffers with a common access control list (ACL), i.e., a set of processes with access to all IO-Lite buffers in the pool. The choice of a pool from which a new IO-Lite buffer is allocated determines the ACL of the data stored in the buffer.

IO-Lite's access control model requires programs to determine the ACL of an I/O data object prior to storing it in main memory, in order to avoid copying. Determining the ACL is trivial in most cases, except when an incoming packet arrives at a network interface, as discussed in Section 2.6.

Figure 1 depicts the relationship between VM pages, buffers, and buffer aggregates. IO-Lite buffers are allocated in a region of the virtual address space called the *IO-Lite window*. The IO-Lite window appears in the virtual address spaces of all protection domains, including the kernel. The figure shows a section of the IO-Lite window populated by three buffers. An IO-Lite buffer always consists of an integral number of (virtually) contiguous VM pages. The pages of an IO-Lite buffer share identical access control attributes; that is, in a given protection domain, either all or none of a buffer's pages are accessible.

Also shown are two buffer aggregates. An aggregate contains an ordered list of tuples of the form <*address, length*>. Each tuple refers to a subrange of memory called a *slice*. A slice is always contained

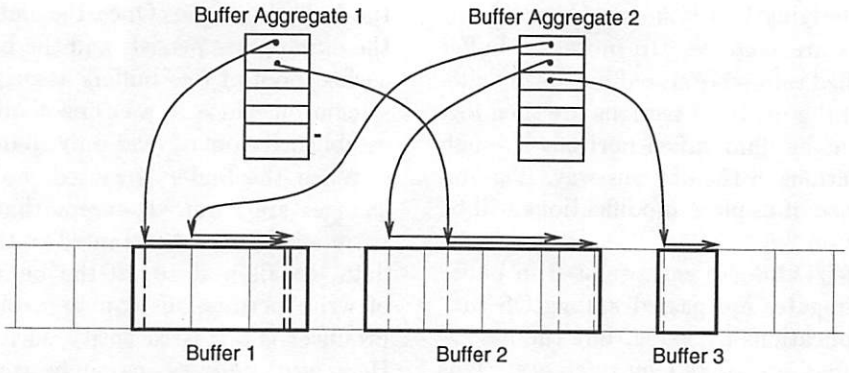


Figure 1: Aggregate buffers and slices

in one IO-Lite buffer, but slices in the same IO-Lite buffer may overlap. The contents of a buffer aggregate can be enumerated by reading the contents of each of its constituent slices in order.

Data objects with the same ACL can be allocated in the same IO-Lite buffer and on the same page. As a result, IO-Lite does not waste memory when allocating objects that are smaller than the VM page size.

2.4 IO-Lite and Applications

To take *full* advantage of IO-Lite, application programs can use an extended I/O application programming interface (API) that is based on buffer aggregates. This section briefly describes this API. A complete discussion of the API can be found in the technical report [20].

```
size_t IOL_read(int fd, IOL_Agg **aggr,
                size_t size);
size_t IOL_write(int fd, IOL_Agg *aggr);
```

Figure 2: IO-Lite I/O API

IOL_read and IOL_write form the core of the interface (see Figure 2). These operations supersede the standard UNIX `read` and `write` operations. (The latter operations are maintained for backward compatibility.) Like their predecessors, the new operations can act on any UNIX file descriptor. All other file descriptor related UNIX systems calls remain unchanged.

The new IOL_read operation returns a buffer aggregate (IOL_Agg) containing at most the amount of data specified as an argument. Unlike the POSIX read, IOL_read may always return less data than requested. The IOL_write operation replaces the data in an external data object with the contents of the buffer aggregate passed as an argument.

The effects of IOL_read and IOL_write operations are *atomic* with respect to other IOL_write operations concurrently invoked on the same descriptor. That is, an IOL_read operations yields data that either reflects all or none of the changes affected by a concurrent IOL_write operation on the same file descriptor. The data returned by a IOL_read is effectively a "snapshot" of the data contained in the object associated with the file descriptor.

Additional IO-Lite system calls allow the creation and deletion of IO-Lite allocation pools. A version of IOL_read allows applications to specify an allocation pool, such that the system places the requested data into IO-Lite buffers from that pool. Applications that manage multiple I/O streams with different access control lists use this operation. The (IOL_Agg) buffer aggregate abstract data type supports a number of operations for creation, destruction, duplication, concatenation and truncation as well as data access.

Implementations of language-specific runtime I/O libraries, like the ANSI C *stdio* library, can be converted to use the new API internally. Doing so reduces data copying without changing the library's API. As a result, applications that perform I/O using these standard libraries can enjoy some performance benefits merely by re-linking them with the new library.

2.5 IO-Lite and the Filesystem

With IO-Lite, buffer aggregates form the basis of the filesystem cache. The filesystem itself remains unchanged.

File data that originates from a local disk is generally page-aligned and page sized. However, file data received from the network may not be page-aligned or page-sized, but can nevertheless be kept in the file cache without copying. Conventional UNIX file cache implementations are not suitable for IO-Lite, since they place restrictions on the layout of cached file data. As a result, current Unix implementations perform a copy when file data arrives from the network.

The IO-Lite file cache has no statically allocated storage. The data resides in IO-Lite buffers, which occupy ordinary pageable virtual memory. Conceptually, the IO-Lite file cache is very simple. It consists of a data structure that maps triples of the form <*file-id, offset, length*> to buffer aggregates that contain the corresponding extent of file data.

Since IO-Lite buffers are immutable, a write operation to a cached file results in the replacement of the corresponding buffers in the cache with the buffers supplied in the write operation. The replaced buffers no longer appear in the file cache; however, they persist as long as other references to them exist.

For example, assume that a IOL_read operation of a cached file is followed by a IOL_write operation to the same portion of the file. The buffers that were returned in the IOL_read are replaced in the cache as a result of the IOL_write. However, the buffers persist until the process that called IOL_read deallocates them and no other references to the buffers remain. In this way, the snapshot semantics of the IOL_read operation are preserved.

2.6 IO-Lite and the Network

With IO-Lite, the network subsystem uses IO-Lite buffer aggregates to store and manipulate network packets.

Some modifications are required to network device drivers. As explained in Section 2.2, programs using IO-Lite must determine the ACL of a data object prior to storing the object in memory. Thus, network interface drivers must determine the I/O stream associated with an incoming packet, since this stream implies the ACL for the data contained in the packet.

To avoid copying, drivers must determine this information from the headers of incoming packets using a packet filter [16], an operation known as *early demultiplexing*. Incidentally, early demultiplexing has been identified by many researchers as a necessary feature for efficiency and quality of service in high-performance networks [23]. With IO-Lite, early demultiplexing is necessary for best performance.

2.7 Cache Replacement and Paging

We now discuss the mechanisms and policies for managing the IO-Lite file cache and the physical memory used to support IO-Lite buffers. There are two related issues, namely (1) replacement of file cache entries, and (2) paging of virtual memory pages that contain IO-Lite buffers. Since cached file data resides in IO-Lite buffers, the two issues are closely related.

Cache replacement in a unified caching/buffering system is different from that of a conventional file cache. Cached data is potentially concurrently accessed by applications. Therefore, replacement decisions should take into account both references to a cache entry (i.e., IOL_read and IOL_write operations), as well as virtual memory accesses to the buffers associated with the entry[3].

Moreover, the data in an IO-Lite buffer can be shared in complex ways. For instance, assume that an application reads a data record from file A, appends that record to the same file A, then writes the record to a second file B, and finally transmits the record via a network connection. After this sequence of operations, the buffer containing the record will appear in two different cache entries associated with file A (corresponding to the offset from where the record was read, and the offset at which it was appended), in a cache entry associated with file B, in the network subsystem transmission buffers, and in the user address space of the application. In general, the data in an IO-Lite buffer may at the same time be part of an application data structure, represent buffered data in various OS subsystems, and represent cached portions of several files or different portions of the same file.

Due to the complex sharing relationships, a large design space exists for cache replacement and paging of unified I/O buffers. While we expect that further research is necessary to determine the best policies, our current system employs the following simple strategy. Cache entries are maintained in a list ordered first by current use (i.e, is the data currently referenced by anything other than the cache?), then by time of last access, taking into account read and write operations but not VM accesses for efficiency. When a cache entry needs to be evicted, the least recently used among currently not referenced cache entries is chosen, else the least recently used among the currently referenced entries.

Cache entry eviction is triggered by a simple rule that is evaluated each time a VM page containing cached I/O data is selected for replacement by the VM pageout daemon. If, during the period since the last cache entry eviction, more then half of VM pages selected for replacement were pages containing cached I/O data, then it is assumed that the current file cache is too large, and we evict one cache entry. Because the cache is enlarged (i.e., a new entry is added) on every miss in the file cache, this policy tends to keep the file cache at a size such that about half of all VM page replacements affect file cache pages.

Since all IO-Lite buffers reside in pageable virtual

[3] Similar issues arise in file caches that are based on memory mapped files.

memory, the cache replacement policy only controls how much data the file cache *attempts* to hold. Actual assignment of physical memory is ultimately controlled by the VM system. When the VM pageout daemon selects a IO-Lite buffer page for replacement, IO-Lite writes the page's contents to the appropriate backing store and frees the page.

Due to the complex sharing relationships possible in a unified buffering/caching system, the contents of a page associated with a IO-Lite buffer may have to be written to multiple backing stores. Such backing stores include ordinary paging space, plus one or more files for which the evicted page is holding cached data.

Finally, IO-Lite includes support for application-specific file cache replacement policies. Interested applications can customize the policy using an approach similar to that proposed by Cao et al. [9].

2.8 Impact of Immutable I/O buffers

Consider the impact of IO-Lite's immutable I/O buffers on program operation. If a program wishes to modify a data object stored in a buffer aggregate, it must store the new values in a newly allocated buffer. There are three cases to consider.

First, if every word in the data object is modified, then the only additional cost (over in-place modification) is a buffer allocation. This case arises frequently in programs that perform operations such as compression and encryption. The absence of support for in-place modifications should not significantly affect the performance of such programs.

Second, if only a subset of the words in the object change values, then the naive approach of copying the entire object would result in partially redundant copying. This copying can be avoided by storing modified values into a new buffer, and logically combining (chaining) the unmodified and modified portions of the data object through the operations provided by the buffer aggregate.

The additional costs in this case (over in-place modification) are due to buffer allocations and chaining (during the modification of the aggregate), and subsequent increased indexing costs (during access of the aggregate) incurred by the non-contiguous storage layout. This case arises in network protocols (fragmentation/reassembly, header addition/removal), and many other programs that reformat/reblock I/O data units. The performance impact on these programs due to the lack of in-place modification is small as long as changes to data objects are reasonably localized.

The third case arises when the modifications of the data object are so widely scattered (leading to a highly fragmented buffer aggregate) that the costs of chaining and indexing exceed the cost of a redundant copy of the entire object into a new, contiguous buffer. This case arises in many scientific applications that read large matrices from input devices and access/modify the data in complex ways. For such applications, contiguous storage and in-place modification is a must. For this purpose, IO-Lite incorporates the *mmap* interface found in all modern UNIX systems. The mmap interface creates a contiguous memory mapping of an I/O object that can be modified in-place.

The use of mmap may require copying in the kernel. First, if the data object is not contiguous and not properly aligned (e.g. incoming network data) a copy operation is necessary due to hardware constraints. In practice, the copy operation is done lazily on a per-page basis. When the first access occurs to a page of a memory mapped file, and its data is not properly aligned, that page is copied.

Second, a copy is needed in the event of a store operation to a memory-mapped file, when the affected page is also referenced through an immutable IO-Lite buffer. (This case arises, for instance, when the file was previously read by some user process using an IOL_read operation). The modified page must be copied in order to maintain the snapshot semantics of the IOL_read operation. The copy is performed lazily, upon the first write access to a page.

2.9 Cross-Subsystem Optimizations

A unified buffering/caching system enables certain optimizations across applications and OS subsystems not possible in conventional I/O systems. These optimizations leverage the ability to uniquely identify a particular I/O data object throughout the system.

For example, with IO-Lite, the Internet checksum module used by the TCP and UDP protocols is equipped with an optimization that allows it to cache the Internet checksum computed for each slice of a buffer aggregate. Should the same slice be transmitted again, the cached checksum can be reused, avoiding the expense of a repeated checksum calculation. This optimization works extremely well for network servers that serve documents stored on disk with a high degree of locality. Whenever a file is requested that is still in the IO-Lite file cache, TCP can reuse a precomputed checksum, thereby eliminating the only remaining data-touching operation on the critical I/O path.

To support such optimizations, IO-Lite provides with each buffer a *generation number*. The generation number is incremented every time a buffer is

re-allocated. Since IO-Lite buffers are immutable, this generation number, combined with the buffer's address, provides a system-wide unique identifier for the *contents* of the buffer. That is, when a subsystem is presented repeatedly with an IO-Lite buffer with an identical address and an identical generation number, it can be sure that the buffer contains the same data values, thus enabling optimizations like Internet checksum caching.

2.10 Operation in a Web Server

In this section, we describe the operation of IO-Lite in a Web server as an example. We start with an overview of the basic operation of a Web server on a conventional UNIX system.

A Web server repeatedly accepts TCP connections from clients, reads the client's HTTP request, and transmits the requested content data with an HTTP response header. If the requested content is static, the corresponding document is read from the file system. If the document is not found in the filesystem's cache, a disk read is necessary.

Copying occurs as part of the reading of data from the filesystem, and when the data is written to the socket attached to the client's TCP connection. High-performance Web servers avoid the first copy by using the UNIX mmap interface to read files, but the second copy remains. Multiple buffering occurs because a given document may simultaneously be stored in the file cache and in the TCP retransmission buffers of potentially multiple client connections.

With IO-Lite, all data copying and multiple buffering is eliminated. Once a document is in main memory, it can be served repeatedly by passing buffer aggregates between the file cache, the server application, and the network subsystem. The server obtains a buffer aggregate using the IOL_read operation on the appropriate file descriptor, concatenates a response header, and transmits the resulting aggregate using IOL_write on the TCP socket. If a document is served repeatedly from the file cache, the TCP checksum need not be recalculated except for the buffer containing the response header.

Dynamic content is typically generated by an auxiliary third-party CGI program that runs as a separate process. The data is sent from the CGI process to the server process via a UNIX pipe. In conventional systems, sending data across the pipe involves at least one data copy. In addition, many CGI programs read primary files that they use to synthesize dynamic content from the filesystem, causing more data copying when that data is read. Caching of dynamic content in a CGI program can aggravate the multiple buffering problem: Primary files used to synthesize dynamic content may now be stored in the file cache, in the CGI program's cache as part of a dynamic page, in the server's holding buffers, and in the TCP retransmission buffers.

With IO-Lite, sending data over a pipe involves no copying. CGI programs can synthesize dynamic content by manipulating buffer aggregates containing data from primary files and newly generated data. Again, IO-Lite eliminates all copying and multiple buffering, even in the presence of caching CGI programs. TCP checksums need not be recomputed for portions of dynamically generated content that are repeatedly transmitted.

IO-Lite's ability to eliminate data copying and multiple buffering can dramatically reduce the cost of serving static and dynamic content. The impact is particularly strong in the case when a cached copy (static or dynamic) of the requested content exists, since copying costs can dominate the service time in this case. Moreover, the elimination of multiple buffering frees up valuable memory resources, permitting a larger file cache size and hit rate, thus further increasing server performance.

Finally, a Web server can use the IO-Lite facilities to customize the replacement policy used in the file cache to derive further performance benefits. To use IO-Lite, an existing Web server need only be modified to use the IO-Lite API. CGI programs must likewise use buffer aggregates to synthesize dynamic content.

A quantitative evaluation of IO-Lite in the context of a Web server follows in Section 4.

3 Implementation

This section gives an overview of the implementation of the IO-Lite prototype system in a 4.4BSD derived UNIX kernel. IO-Lite is implemented as a loadable kernel module that can be dynamically linked to a slightly modified FreeBSD 2.2.6 kernel. A runtime library must be linked with applications wishing to use the IO-Lite API. This library provides the buffer aggregate manipulation routines and stubs for the IO-Lite system calls.

Network Subsystem: The BSD network subsystem was adapted by encapsulating IO-Lite buffers inside the BSD native buffer abstraction, mbufs. This approach avoids intrusive and widespread source code modifications.

The encapsulation was accomplished by using the mbuf out-of-line pointer to refer to an IO-Lite buffer, thus maintaining compatibility with the BSD network subsystem in a very simple, efficient manner. Small data items such as network packet headers

are still stored inline in mbufs, but the performance critical bulk data resides in IO-Lite buffers. Since the mbuf data structure remains essentially unmodified, the bulk of the network subsystem (including all network protocols) works unmodified with mbuf encapsulated IO-Lite buffers.

Filesystem: The IO-Lite file cache module replaces the unified buffer cache module found in 4.4BSD derived systems [17]. The bulk of the filesystem code (below the block-oriented file read/write interface) remains unmodified. As in the original BSD kernel, the filesystem continues to use the "old" buffer cache to hold filesystem metadata.

The original UNIX read and write system calls for files are implemented by IO-Lite for backward compatibility; a data copy operation is used to move data between application buffers and IO-Lite buffers.

VM System: Adding IO-Lite does not require any significant changes to the BSD VM system [17]. IO-Lite uses standard interfaces exported by the VM system to create a VM object that represents the IO-Lite window, map that object into kernel and user process address spaces, and to provide page-in and page-out handlers for the IO-Lite buffers.

The page-in and page-out handlers use information maintained by the IO-Lite file cache module to determine the disk locations that provide backing store for a given IO-Lite buffer page. The replacement policy for IO-Lite buffers and the IO-Lite file cache is implemented by the page-out handler, in cooperation with the IO-Lite file cache module.

IPC System: The IO-Lite system adds a modified implementation of the BSD IPC facilities. This implementation is used whenever a process uses the IO-Lite read/write operations on a BSD pipe or Unix domain socket. If the processes on both ends of a pipe or Unix domain socket-pair use the IO-Lite API, then the data transfer proceeds copy-free by passing the associated IO-Lite buffers by reference. The IO-Lite system ensures that all pages occupied by these IO-Lite buffers are readable in the receiving domain, using standard VM operations.

4 Performance

In this section, we evaluate the performance of a prototype IO-Lite implementation. All experiments use a server system based on a 333MHz Pentium II PC equipped with 128MB of main memory and five network adaptors to a switched 100Mbps Fast Ethernet.

To fully expose the performance bottlenecks in the operating system, we use a high-performance in-house Web server called *Flash* [19]. Flash is an event-driven HTTP server with support for CGI. To the best of our knowledge, Flash is among the fastest HTTP servers currently available. *Flash-Lite* is a slightly modified version of Flash that uses the IO-Lite API.

While Flash uses memory-mapped files to read disk data, Flash-Lite uses the IO-Lite read/write interface to access disk files. In addition, Flash-Lite uses the IO-Lite support for customization of the file caching policy to implement Greedy Dual Size (GDS), a policy that performs well on Web workloads [10].

For comparison, we also present performance results with Apache version 1.3.1, a widely used Web server [3]. This version uses mmap to read files and performs substantially better than earlier versions. Apache's performance reflects what can be expected of a widely used Web server today.

Flash is an aggressively optimized, experimental Web server; it reflects the best in Web server performance that can be achieved using the standard facilities available in a modern operating system. Flash-Lite's performance reflects the additional benefits that result from IO-Lite. All Web servers were configured to use a TCP socket send buffer size of 64KBytes; access logging was disabled.

In the first experiment, 40 HTTP clients running on five machines repeatedly request the same document of a given size from the server. A client issues a new request as soon as a response is received for the previous request [4]. The file size requested varies from 500 bytes to 200KBytes (the data points below 20KB are 500 bytes, 1KB, 2KB, 3KB, 5KB, 7KB, 10KB and 15 KB). In all cases, the files are cached in the server's file cache after the first request, so no physical disk I/O occurs in the common case.

Figure 3 shows the output bandwidth of the various Web servers as a function of request file size. Results are shown for Flash-Lite, Flash and Apache. Flash performs consistently better than Apache, with bandwidth improvements up to 71% at a file size of 20KBytes. This result confirms that our aggressive Flash server outperforms the already fast Apache server.

Flash using IO-Lite (Flash-Lite) delivers a bandwidth increase of up to 43% over Flash and up to 137% over Apache. For file sizes of 5KBytes or less, Flash and Flash-Lite perform equally well. The reason is that at these small sizes, control overheads, rather than data dependent costs, dominate the cost of serving a request.

The throughput advantage obtained with IO-Lite in this experiment reflects only the savings due to

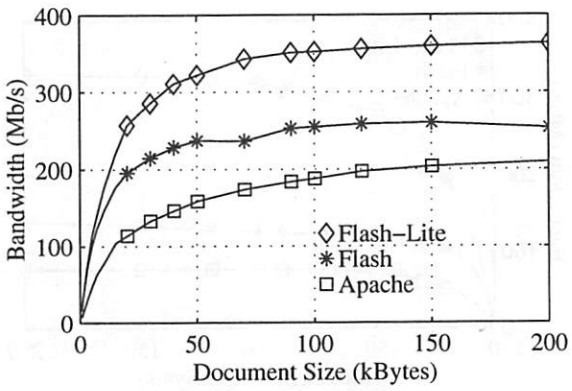


Figure 3: HTTP

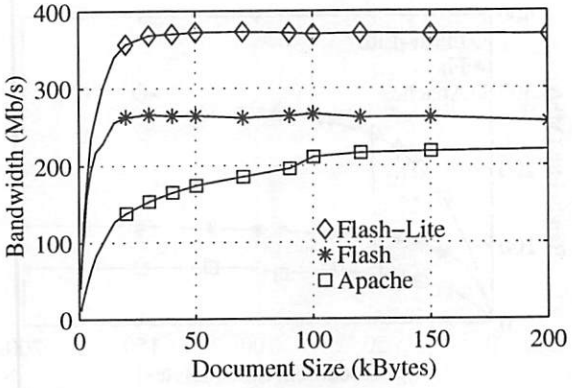


Figure 4: Persistent HTTP

copy-avoidance and checksum caching. Potential benefits resulting from the elimination of multiple buffering and the customized file cache replacement are not realized, because this experiment does not stress the file cache (i.e., a single document is repeatedly requested).

4.1 Persistent Connections

The previous experiments are based on HTTP 1.0, where a TCP connection is established by clients for each individual request. The HTTP 1.1 specification adds support for persistent (keep-alive) connections that can be used by clients to issue multiple requests in sequence. We modified both versions of Flash to support persistent connections and repeated the previous experiment. The results are shown in Figure 4.

With persistent connections, the request rate for small files (less than 50KBytes) increases significantly with Flash and Flash-Lite, due to the reduced overhead associated with TCP connection establishment and termination. The overheads of the process-per-connection model in Apache appear to prevent that server from fully taking advantage of this effect.

Persistent connections allow Flash-Lite to realize its full performance advantage over Flash at smaller file sizes. For files of 20KBytes and above, Flash-Lite outperforms Flash by up to 43%. Moreover, Flash-Lite comes within 10% of saturating the network at a file size of only 17KBytes and it saturates the network for file sizes of 30KBytes and above.

4.2 CGI Programs

An area where IO-Lite promises particularly substantial benefits is CGI programs. When compared to the original CGI 1.1 standard [1], the newer FastCGI interface [2] amortizes the cost of forking and starting a CGI process by allowing such processes to persist across requests. However, there are still substantial overheads associated with IPC across pipes and multiple buffering, as explained in Section 2.10.

We performed an experiment to evaluate how IO-Lite affects the performance of dynamic content generation using FastCGI programs. A test CGI program, when receiving a request, sends a "dynamic" document of a given size from its memory to the server process via the UNIX pipe; the server transmits the data on the client's connection. The results of these experiments are shown in Figure 5.

The bandwidth of the Flash and Apache servers is roughly half their corresponding bandwidth on static documents. This results shows the strong impact of the copy-based pipe IPC in regular UNIX on CGI performance. With Flash-Lite, the performance is significantly better, approaching 87% of the speed on static content. Also interesting is that CGI programs with Flash-Lite achieve performance better than static files with Flash.

Figure 6 shows results of the same experiment using persistent HTTP-1.1 connections. Unlike Flash-Lite, Flash and Apache cannot take advantage of the efficiency of persistent connections here, since their performance is limited by the pipe IPC.

The results of these experiments show that IO-Lite allows a server to efficiently support dynamic content using CGI programs, without giving up fault isolation and protection from such third-party programs. This result suggests that with IO-Lite, there may be less reason to resort to library-based interfaces for dynamic content generation. Such interfaces were defined by Netscape and Microsoft [18, 14] to avoid the overhead of CGI. Since they require third-party programs to be linked with the server, they give up fault isolation and protection.

4.3 Performance on Real Workload

The previous experiments use an artificial workload. In particular, they use a set of requested documents that fits into the server's main memory cache.

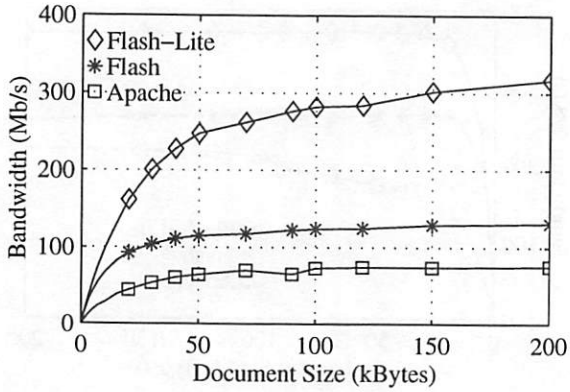


Figure 5: HTTP/FastCGI

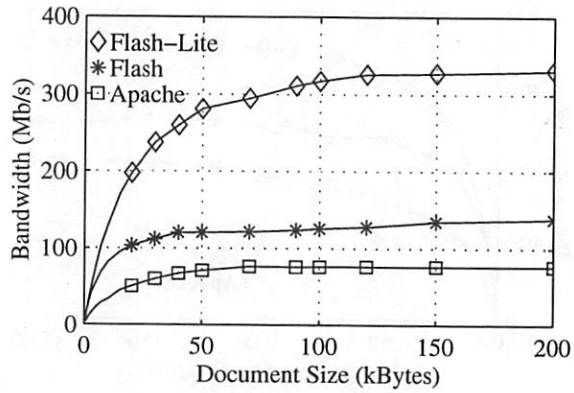


Figure 6: P-HTTP/FastCGI

	Apache	Flash	Flash-Lite
Requests/sec	524	617	866
Ratio	1.0	1.18	1.65

Table 1: Rice Trace

As a result, these experiments only quantify the increase in performance due to the elimination of CPU overhead with IO-Lite. They do not demonstrate possible secondary performance benefits due to the increased availability of main memory that results from IO-Lite's elimination of double buffering. Increasing the amount of available memory allows a larger set of documents to be cached, thus increasing the server cache hit rate and performance. Finally, since the cache is not stressed in these experiments, possible performance benefits due to the customized file cache replacement policy used in Flash-Lite are not exposed.

To measure the overall impact of IO-Lite on the performance of a Web server under realistic workload conditions, we performed experiments where our experimental server is driven by a workload derived from server logs of an actual Web server. We use logs from Rice University's Computer Science departmental Web server. Only requests for static documents were extracted from the logs. The average request size in this trace is about 17KBytes.

Table 1 show the relative performance in requests/sec of Flash-Lite, Flash, and Apache on the Rice CS department trace. Flash exceeds the throughput of Apache by 18% on this trace. Flash-Lite gains 65% throughput over Apache and 40% over Flash, demonstrating the effectiveness of IO-Lite under realistic workload conditions, where the set of requested documents exceeds the cache size and disk accesses occur.

4.4 WAN Effects

Our experimental testbed uses a local-area network to connect a relatively small number of clients to the experimental server. This setup leaves a significant aspect of real Web server performance unevaluated, namely the impact of wide-area network delays and large numbers of clients [4]. In particular, we are interested here in the TCP retransmission buffers needed to support efficient communication on connections with substantial bandwidth-delay products.

Since both Apache and Flash use mmap to read files, the remaining primary source of double buffering is TCP's transmission buffers. The amount of memory consumed by these buffers is related to the number of concurrent connections handled by the server, times the socket send buffer size T_{ss} used by the server. For good network performance, T_{ss} must be large enough to accommodate a connection's bandwidth-delay product. A typical setting for T_{ss} in a server today is 64KBytes.

Busy servers may handle several hundred concurrent connections, resulting in significant memory requirements even in the current Internet. With future increases in Internet bandwidth, the necessary T_{ss} settings needed for good network performance are likely to increase significantly, which makes it increasingly important to eliminate double buffering.

The BSD UNIX network subsystem dynamically allocates mbufs (and mbuf clusters) to hold data in socket buffers. When the server is contacted by a large number of clients concurrently and the server transmits on each connection an amount of data equal or larger than T_{ss}, then the system may be forced to allocate sufficient mbufs to hold T_{ss} bytes for each connection. Moreover, in FreeBSD and other BSD-derived system, the size of the mbuf pools is never decreased. That is, once the mbuf pool has grown to a certain size, its memory is permanently unavailable for other uses, such as the file cache.

To quantify this effect, we repeated the previous

experiment, with the addition that an increasing number of "slow" background clients contact the server. These clients request a document, but are slow to read the data from their end of the TCP connection, which has a small receive buffer (2KB). This trick forces the server to buffer data in its socket send buffers and simulates the effect of WAN connections on the server.

As the number of clients increases, more memory is used to hold data in the server's socket buffers, increasing memory pressure and reducing the size of the file cache. With IO-Lite, however, socket send buffers do not require separate memory since they refer to data stored in IO-Lite buffers[4]. Double buffering is eliminated, and the amount of memory available for the file cache remains independent of the number of concurrent clients contacting the server and the setting of T_{ss}.

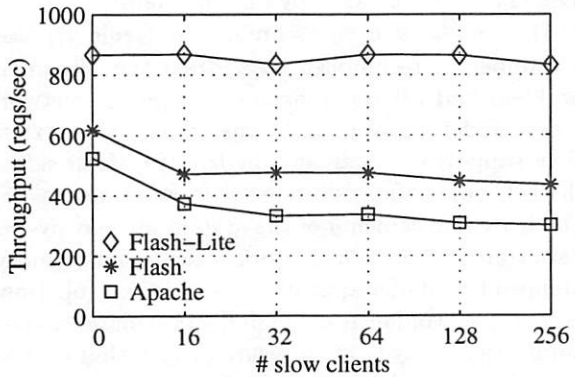


Figure 7: Throughput vs. #clients

Figure 7 shows the performance of Apache, Flash and Flash-Lite as a function of the number of slow clients contacting the server. As expected, Flash-Lite remains unaffected by the number of slow clients contacting the server, up to experimental noise. Apache suffers up to 42% and Flash up to 30% throughput loss as the number of clients increases, reducing the available cache size. For 16 slow clients and more, Flash-Lite is close to 80% faster than Flash; for 32 slow clients and more, Flash-Lite is 150% faster than Apache.

The results confirm IO-Lite's ability to eliminate double buffering in the network subsystem. This effect gains in importance both as the number of concurrent clients and the setting of T_{ss} increases. Future increases in Internet bandwidth will require larger T_{ss} settings to achieve good network utilization.

[4] A small amount of memory is required to hold mbuf structures.

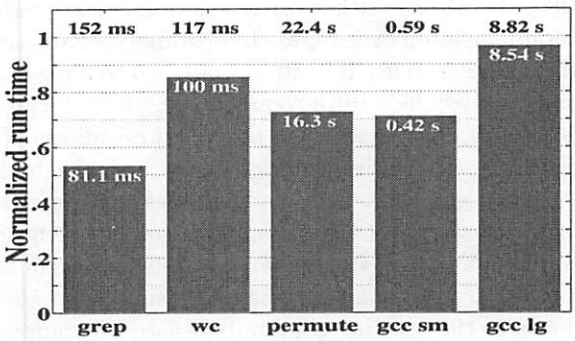


Figure 8: Various application runtimes

4.5 Other Applications

To demonstrate the impact of IO-Lite on the performance of a wider range of applications, and also to gain experience with the use of the IO-Lite API, a number of existing UNIX programs were converted and some new programs were written to use IO-Lite. We modified GNU grep, wc, cat, and the GNU gcc compiler chain (compiler driver, C preprocessor, C compiler, and assembler). Figure 8 depicts the results obtained with grep, wc, and permute. The "wc" refers to a run of the word-count program on a 1.75 MB file. The file is in the file cache, so no physical I/O occurs. "Permute" generates all possible permutations of 8 character words in a 80 character string. Its output ($10! * 80 = 290304000$ bytes) is piped into the "wc" program. The "grep" bar refers to a run of the GNU grep program on the same file used for the "wc" program, but the file is piped into wc from cat instead of being read directly from disk.

Improvements in runtime of approximately 15% result from the use of IO-Lite for wc, since it reads cached files. The benefit of IO-Lite is reduced because each page of the cached file must be mapped into the application's address space when a file is read from the IO-Lite file cache.

For the "permute" program the improvement is more significant (24%). The reason is that a pipeline is involved in the latter program. Whenever local interprocess communication occurs, the IO-Lite implementation can recycle buffers, avoiding all VM map operations. Finally, in the "grep" case, the overhead of multiple copies is eliminated, so the IO-Lite version is able to eliminate 3 copies (one due to "grep", and two due to "cat").

The gcc compiler chain was converted mainly to determine if there were benefits from IO-Lite for more compute-bound applications and to stress the IO-Lite implementation. We expected that a compiler is too compute-intensive to benefit substantially from I/O performance improvements. Rather than modify the entire program, we simply replaced

the stdio library with a version that uses IO-Lite for communication over pipes. Interestingly, converting the compiler to use IO-Lite actually led to a measurable performance improvement. The improvement is mainly due to the fact that IO-Lite allows efficient communication through pipes. Although the standard gcc has an option that uses pipes instead of temporary files for communication between the compiler's various stages, various inefficiencies in the handling of pipes actually caused a significant slowdown, so the baseline gcc numbers used for comparison are for gcc running without pipes. Since IO-Lite can handle pipes very efficiently, unexpected performance improvements resulted from its use. The "gcc sm" and "gcc lg" bars refer to compiles of a 1200 Byte and a 206 KByte file, respectively.

The "grep" and "wc" programs read their input sequentially, and were converted to use the IO-Lite API. The C preprocessor's output, the compiler's input and output, and the assembler's input all use the C stdio library, and were converted merely by relinking them with an IO-Lite version of stdio library. The preprocessor (cpp) uses mmap to read its input.

5 Related Work

This section discusses related work. To provide focus, we examine how existing and proposed I/O systems affect the design and performance of a Web server. We begin with the standard UNIX (POSIX) I/O interface, and go on to more aggressively optimized I/O systems proposed in the literature.

POSIX I/O: The UNIX/POSIX read/readv operations allow an application to request the placement of input data at an arbitrary (set of) location(s) in its private address space. Furthermore, both the read/readv and write/writev operations have copy semantics, implying that applications can modify data that was read/written from/to an external data object without affecting that data object.

To avoid the copying associated with reading a file repeatedly from the filesystem, a Web server using this interface would have to maintain a user-level cache of Web documents, leading to double-buffering in the disk cache and the server. When serving a request, data is copied into socket buffers, creating a third copy. CGI programs [1] cause data to be additionally copied from the CGI program into the server's buffers via a pipe, possibly involving kernel buffers.

Memory-mapped files: The semantics of mmap facilitate a copy-free implementation, but the contiguous mapping requirement may still demand copying in the OS for data that arrives from the network. Like IO-Lite, mmap avoids multiple buffering of file data in file cache and application(s). Unlike IO-Lite, mmap does not generalize to network I/O, so double buffering (and copying) still occurs in the network subsystem.

Moreover, memory-mapped files do not provide a convenient method for implementing CGI support, since they lack support for producer/consumer synchronization between the CGI program and the server. Having the server and the CGI program share memory-mapped files for IPC requires ad-hoc synchronization and adds complexity.

Transparent Copy Avoidance: In principle, copy avoidance and single buffering could be accomplished transparently using existing POSIX APIs, through the use of page remapping and copy-on-write. Well-known difficulties with this approach are VM page alignment problems, and potential writes to buffers by applications, which may defeat copy avoidance by causing copy-on-write faults.

The *emulated copy* technique in Genie [7] uses a number of techniques to address the alignment problem and allows transparent copy-free network access under certain conditions. Subsequent extensions support transparent copy-free IPC if one side of the IPC connection is a trusted (server) process [8]. Further enhancements of the system allow copy-free data transfer between network sockets and memory-mapped files under appropriate conditions [6]. However, copy avoidance is not fully transparent, since applications may have to ensure proper alignment of incoming network data, use buffers carefully to avoid copy-on-write faults, and use special system calls to move data into memory-mapped files.

To use Genie in a Web server, for instance, the server application must be modified to use memory-mapped files and to satisfy other conditions necessary to avoid copying. Due to the lack of support for copy-free IPC between unprivileged processes in Genie, CGI applications may require data copying.

IO-Lite does not attempt to provide transparent copy avoidance. Instead, I/O-intensive applications must be written or modified to use the IO-Lite API. (Legacy applications with less stringent performance requirements can be supported in a backward-compatible fashion at the cost of a copy operation, as in conventional systems.) By giving up transparency and in-place modifications, IO-Lite can support universal copy-free I/O, including general IPC and cached file access, using an API with simple semantics and consistent performance.

Copy Avoidance with Handoff Semantics: The *Container Shipping* (CS) I/O system [21] and Thadani and Khalidi's work [24] use I/O read and

write operations with handoff (move) semantics. Like IO-Lite, these systems require applications to process I/O data at a given location. Unlike IO-Lite, they allow applications to modify I/O buffers in-place. This is safe because the handoff semantics permit only sequential sharing of I/O data buffers—i.e., only one protection domain has access to a given buffer at any time.

Sacrificing concurrent sharing comes at a cost: Since an application loses access to a buffer that it passed as an argument to a write operation, an explicit physical copy is necessary if the application needs access to the data after the write. Moreover, when an application reads from a file while a second application is holding cached buffers for the same file, a second copy of the data must be read from the input device. This scenario demonstrates that the lack of support for concurrent sharing prevents an effective integration of a copy-free I/O buffering scheme with the file cache.

In a Web server, lack of concurrent sharing requires copying of "hot" pages, making the common case more expensive. CGI programs that produce entirely new data for every request (as opposed to returning part of a file or a set of files) are not affected, but CGI programs that try to intelligently cache data suffer copying costs.

Fbufs: Fbufs is a copy-free cross-domain transfer and buffering mechanism for I/O data, based on immutable buffers that can be concurrently shared. The fbufs system was designed primarily for handling network streams, was implemented in a non-UNIX environment, and does not support filesystem access or a file cache. IO-Lite's cross-domain transfer mechanism was inspired by fbufs. When trying to use fbufs in a Web server, the lack of integration with the filesystem would result in double-buffering. Their use as an interprocess communication facility would benefit CGI programs, but with the same restrictions on filesystem access.

Extensible Kernels: Recent work has proposed the use of of *extensible* kernels [5, 12, 15, 22] to address a variety of problems associated with existing operating systems. Extensible kernels can potentially address many different OS performance problems, not just the I/O bottleneck that is the focus of our work.

In contrast to extensible kernels, IO-Lite is directly applicable to existing general-purpose operating systems and provides an application-independent scheme for addressing the I/O bottleneck. Our approach avoids the complexity and the overhead of new safety provisions required by extensible kernels. It also relieves the implementors of servers and applications from having to write OS-specific kernel extensions.

CGI programs may pose problems for extensible kernel-based Web servers, since some protection mechanism must be used to insulate the server from poorly-behaved programs. Conventional Web servers and Flash-Lite rely on the operating system to provide protection between the CGI process and the server, and the server does not extend any trust to the CGI process. As a result, the malicious or inadvertent failure of a CGI program will not affect the server.

To summarize, IO-Lite differs from existing work in its generality, its integration of the file cache, its support for cross-subsystem optimizations, and its direct applicability to general-purpose operating systems. IO-Lite is a general I/O buffering and caching system that avoids all redundant copying and multiple buffering of I/O data, even on complex data paths that involve the file cache, interprocess communication facilities, network subsystem and multiple application processes.

6 Conclusion

This paper presents the design, implementation, and evaluation of IO-Lite, a unified buffering and caching system for general-purpose operating systems. IO-Lite improves the performance of servers and other I/O-intensive applications by eliminating all redundant copying and multiple buffering of I/O data, and by enabling optimizations across subsystems.

Experimental results from a prototype implementation in FreeBSD show performance improvements between 40 and 80% over an already aggressively optimized Web server without IO-Lite, both on synthetic workloads and on real workloads derived from Web server logs. IO-Lite also allows the efficient support of CGI programs without loss of fault isolation and protection. Further results show that IO-Lite reduces memory requirements associated with the support of large numbers of client connections and large bandwidth-delay products in Web servers by eliminating multiple buffering, leading to increased throughput.

Acknowledgments

We are grateful to our OSDI shepherd Greg Minshall and the anonymous reviewers, whose comments have helped to improve this paper. Thanks to Michael Svendsen for his help with the testbed configuration. This work was supported in part by NSF Grants CCR-9803673, CCR-9503098, MIP-9521386,

by Texas TATP Grant 003604, and by an IBM Partnership Award.

References

[1] The common gateway interface. http://hoohoo.ncsa.uiuc.edu/cgi/.

[2] FastCGI specification. http://www.fastcgi.com/.

[3] Apache. http://www.apache.org/.

[4] G. Banga and P. Druschel. Measuring the capacity of a Web server under realistic loads. *World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation)*, 1999. To appear.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. Fifteenth ACM Symp. on Operating System Principles*, Copper Mountain, CO, Dec. 1995.

[6] J. C. Brustoloni. Interoperation of copy avoidance in network and file I/O. In *Proc. of the IEEE Infocom Conference*, New York, Mar. 1999.

[7] J. C. Brustoloni and P. Steenkiste. Effects of buffering semantics on I/O performance. In *Proc. 2nd USENIX Symp. on Operating Systems Design and Implementation*, Seattle WA (USA), Oct. 1996.

[8] J. C. Brustoloni and P. Steenkiste. User-level protocol servers with kernel-level performance. In *Proc. of the IEEE Infocom Conference*, San Francisco, Mar. 1998.

[9] P. Cao and E. Felten. Implementation and performance of application-controlled file caching. In *Proc. of the First USENIX Symp. on Operating System Design and Implementation*, pages 165–177, 1994.

[10] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proc. of the USENIX Symp. on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[11] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proc. of the Fourteenth ACM Symp. on Operating System Principles*, pages 189–202, Dec. 1993.

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the Fifteenth ACM Symp. on Operating System Principles*, Copper Mountain, CO, Dec. 1995.

[13] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[14] Microsoft Corporation ISAPI Overview. http://www.microsoft.com/msdn/sdk/platforms/doc/sdk/internet/src/isapimrg.htm.

[15] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. MacKenzie. Application performance and flexibility on exokernel systems. In *Proc. of the Sixteenth ACM Symp. on Operating System Principles*, San Malo, France, Oct. 1997.

[16] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. of the USENIX '93 Winter Conference*, pages 259–269, Jan. 1993.

[17] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, 1996.

[18] Netscape Server API. http://www.netscape.com/newsref/std/server_api.html.

[19] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server, 1998. Submitted for publication.

[20] V. S. Pai, P. Druschel, and W. Zwaenepoel. I/O-Lite: A unified I/O buffering and caching system. Technical Report 98-331, Department of Computer Science, Rice University, 1998.

[21] J. Pasquale, E. Anderson, and P. K. Muller. Container Shipping: Operating system support for I/O-intensive applications. *IEEE Computer*, 27(3):84–93, Mar. 1994.

[22] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. 2nd USENIX Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[23] D. L. Tennenhouse. Layered multiplexing considered harmful. In H. Rudin and R. Williamson, editors, *Protocols for High-Speed Networks*, pages 143–148, Amsterdam, 1989. North-Holland.

[24] M. N. Thadani and Y. A. Khalidi. An efficient zero-copy I/O framework for UNIX. Technical Report SMLI TR-95-39, Sun Microsystems Laboratories, Inc., May 1995.

Virtual Log Based File Systems for a Programmable Disk*

Randolph Y. Wang† Thomas E. Anderson‡ David A. Patterson†

Abstract

In this paper, we study how to minimize the latency of small synchronous writes to disks. The basic approach is to write to free sectors that are near the current disk head location by leveraging the embedded processor core inside the disk. We develop a number of analytical models to demonstrate the performance potential of this approach. We then present the design of a *virtual log*, a log whose entries are not physically contiguous, and a variation of the log-structured file system based on this approach. The virtual log based file systems can efficiently support small synchronous writes without extra hardware support while retaining the advantages of LFS including its potential to support transactional semantics. We compare our approach against traditional update-in-place and logging systems by modifying the Solaris kernel to serve as a simulation engine. Our evaluations show that random synchronous updates on an unmodified UFS execute up to an order of magnitude faster on a virtual log than on a conventional disk. The virtual log can also significantly improve LFS in cases where delaying small writes is not an option or online cleaning would degrade performance. If the current trends of disk technology continue, we expect the performance advantage of this approach to become even more pronounced in the future.

1 Introduction

In this paper, we set out to answer a simple question: how do we minimize the latency of small synchronous writes to disk?

The performance of small synchronous disk writes impacts the performance of important applications such as recoverable virtual memory [27], persistent object stores [2, 18], and database applications [33, 34]. These systems have become more complex in order to deal with the increasing relative cost of small writes [19].

Similarly, most existing file systems are carefully structured to avoid small synchronous disk writes. UFS by default delays data writes to disk. It is also possible to delay metadata writes if they are carefully ordered [9]. The Log-structured File System (LFS) [25] batches small writes. While the structural integrity of these file systems can be maintained, none of them allows small synchronous writes to be supported efficiently. Write-ahead logging systems [4, 6, 12, 32] accumulate small updates in a log and replay the modifications later by updating in place. Databases often place the log on a separate disk to avoid having the small updates to the log conflict with reads. Our interest is in the limits to small write performance for a single disk.

Because of the limitations imposed by disks, non-volatile RAM (NVRAM) or an uninterruptable power supply (UPS) is often used to provide fast stable writes [3, 14, 15, 19]. However, when write locality exceeds buffer capacity, performance degrades. There are also applications that demand stricter guarantees of reliability and integrity than that of either NVRAM or UPS. Fast small disk writes can provide a cost effective complement to NVRAM.

Our basic approach is to write to a disk location that is close to the head location. We call this *eager writing*. Eager writing requires the file system to be aware of the precise disk head location and disk geometry. One way to satisfy this requirement is to enhance the disk interface to the host so that the host file system can have precise knowledge of the disk state. A second solution is to migrate into the disk some of the file system responsibilities that are traditionally executed on the host. In the rest of this paper, we will assume this second approach, although our techniques do not necessarily depend

*This work was supported in part by the Defense Advanced Research Projects Agency (DABT63-96-C-0056), the National Science Foundation (CDA 9401156), California MICRO, the AT&T Foundation, Digital Equipment Corporation, Hewlett Packard, IBM, Intel, Sun Microsystems, and Xerox Corporation. Anderson was also supported by a National Science Foundation Presidential Faculty Fellowship.

†Computer Science Division, University of California, Berkeley, {rywang,pattrsn}@cs.berkeley.edu

‡Department of Computer Science and Engineering, University of Washington, Seattle, tom@cs.washington.edu

on the ability to run file systems inside disks.

Several technology trends have simultaneously enabled and necessitated the approach of migrating file system responsibility into the disk. First, Moore's Law has driven down the relative cost of CPU power to disk bandwidth, enabling powerful systems to be embedded on disk devices [1]. As this trend continues, it will soon be possible to run the entire file system on the disk. Second, growing at 40% per year [11], disk bandwidth has been scaling faster than other aspects of the disk system. I/O bus performance has been scaling less quickly [21]. The ability of the file system to communicate with the disk (to reorganize the disk, for example) without consuming valuable I/O bus bandwidth has become increasingly important. Disk latency improves even more slowly (at an annual rate of 10% in the past decade [21]). A file system whose small write latency is largely decided by the disk bandwidth instead of any other parameters will continue to perform well. Third, the increasing complexity of the modern disk drives and the fast product cycles make it increasingly difficult for operating system vendors to incorporate useful device heuristics into their file systems to improve performance. By running file system code inside the disk, we can combine the precise knowledge of the file system semantics and detailed disk mechanism to perform optimizations that are otherwise impossible.

The basic concept of performing writes near the disk head position is by no means a new one [5, 8, 10, 13, 23]. But these systems either do not guarantee atomic writes, have poor failure recovery times, or require NVRAM. In this work, we present the design of a *virtual log*, a logging strategy based on eager writing with these unusual features:

- Virtual logging supports fast, synchronous, and atomic disk writes without special hardware support; it serves as a base mechanism upon which efficient transactions can be built.
- The virtual log allows space occupied by obsolete entries to be reused without recopying live entries.
- The virtual log boot straps its recovery from the log tail pointer, which can be stored on disk as part of the firmware power down sequence, allowing efficient normal operations.

We discuss two designs in which the virtual log can be used to improve file system performance. The first is to use it to implement a logical disk interface. This design, called a Virtual Log Disk (VLD), does not alter the existing disk interface and can deliver the performance advantage of eager writing to an unmodified file system. In the second approach, which we have not implemented, we seek a tighter integration of the virtual log into the file system; we present the design of a variation of LFS, called VLFS. We develop analytical models and algorithms to answer a number of fundamental questions about eager writing:

- What is the theoretical limit of its performance?
- How can we ensure open space under the disk head?
- How does this approach fare as different parts of the disk mechanism improve at different rates?

We evaluate our approach against update-in-place and logging by modifying the Solaris kernel to serve as a simulation engine. Our evaluations show that an *unmodified* UFS on an eager writing disk runs about ten times faster than an update-in-place system for small synchronous random updates. Eager writing's economical use of bandwidth also allows it to significantly improve LFS in cases where delaying small writes is not an option or online cleaning would degrade performance. The performance advantage of eager writing should become more profound in the future as technology improves.

Of course, like LFS, these benefits may come at a price of potentially reducing read performance as data may not be optimally placed for future reads. But as increasingly large file caches are employed, modern file systems such as the Network Appliance file system report predominantly write traffic [15]. Large caches also provide opportunity for read reorganization before the reads happen [22].

Although the virtual log shows significant performance promise, this paper remains a preliminary study. A full evaluation would require answers to algorithmic and policy questions of the data reorganizer as part of a complete VLFS implementation. These issues, as well as questions such as how to extend the virtual logging technique to multiple disks, are subjects of our ongoing research.

The remainder of the paper is organized as follows. Section 2 presents the eager writing analytical models. Section 3 presents the design of the virtual log and the virtual log based LFS. Section 4 describes the experimental platform that is used to evaluate the update-in-place, logging, and eager writing strategies. Section 5 shows the experimental results. Section 6 describes some of the related work. Section 7 concludes.

2 Limits to Low Latency Writes

The principle of writing data near the current disk head position is most effective when the head is

always on a free sector. This is not always possible. In this section, we develop a number of analytical models to estimate the amount of time needed to locate free sectors under various utilizations. These models will help us evaluate whether eager writing is a sound strategy, set performance targets for real implementations, and predict future improvements as disks improve. We also use the models to motivate new file system allocation and reorganization algorithms. Because we are interested in the theoretical limits of eager writing latency, the models are for the smallest addressable unit: a disk sector (although the validity of the formulas do not depend on the sector size).

2.1 A Single Track Model

Suppose a disk track contains n sectors, its free space percentage is p, and the free space is randomly distributed. The average number of sectors the disk head must skip before arriving at any free sector is:

$$\frac{(1-p)n}{1+pn} \qquad (1)$$

Appendix A.1 shows a proof of (1) and extends it for situations where the file system block size does not equal the size of the smallest addressable disk unit.

Formula (1) is roughly the ratio between occupied sectors and free ones. This is a promising result for the eager writing approach because, for example, even at a relatively high utilization of 80%, we can expect to incur only a four-sector rotational delay to locate a free sector. For today's disks, this translates to less than 100 μs. In six to seven years, this delay should improve by another order of magnitude because it scales with platter bandwidth. In contrast, it is difficult for an update-in-place system to avoid at least a half-rotation delay. Today, this is, at best, 3 ms, and it improves slowly. This difference is the fundamental reason why eager writing can outperform update-in-place.

2.2 A Single Cylinder Model

We now extend (1) to cover a single cylinder. We compare the time needed to locate the nearest sector in the current track against the time needed to find one in other tracks in the same cylinder and take the minimum of the two. Therefore the expected latency can be expressed as:

$$\sum_x \sum_y \min(x,y) \cdot f_x(p,x) \cdot f_y(p,y) \qquad (2)$$

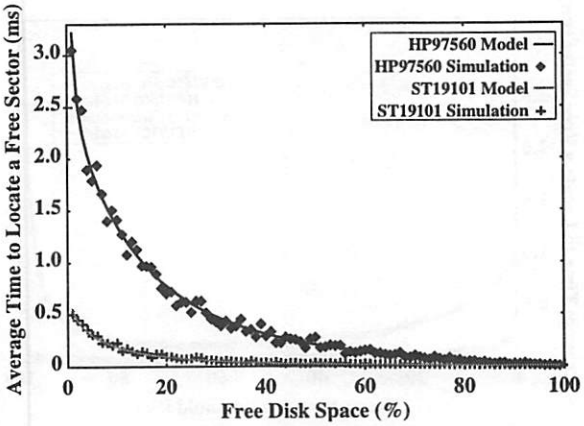


Figure 1: Amount of time to locate the first free sector as a function of the disk utilization.

	HP97560	ST19101
Sectors per Track (n)	72	256
Tracks per Cylinder (t)	19	16
Head Switch (s)	2.5 ms	0.5 ms
Minimum Seek	3.6 ms	0.5 ms
Rotation Speed (RPM)	4002	10000
SCSI Overhead (o)	2.3 ms	0.1 ms

Table 1: Parameters of the HP97560 and the Seagate ST19101 disks.

where x is the delay (in units of sectors) experienced to locate the closest sector in the current track, y is the delay experienced to locate the closest sector in other tracks in the current cylinder, and $f_x(p,x)$ and $f_y(p,y)$ are the probability functions of x and y, respectively, under the assumption of a free space percentage of p. Suppose a head switch costs s and there are t tracks in a cylinder, then the probability functions can be expressed as:

$$f_x(p,x) = p(1-p)^x \qquad (3)$$

$$f_y(p,y) = f_x(1-(1-p)^{t-1}, y-s) \qquad (4)$$

In other words, f_x is the probability that there are x occupied sectors followed by a free sector in the current track, and f_y is the probability that the first $(y-s)$ rotational positions in all $(t-1)$ tracks are occupied and there is one free sector at the next rotational position in at least one of these tracks.

Figure 1 validates the model of (2) with a simulation of the HP97560 and the Seagate ST19101 disks whose relevant parameters are detailed in Table 1. The well validated HP97560 model [17, 26] is approximately eight years old. The state-of-art Seagate disk [20, 28] is more complex but its simulator is only a coarse approximation. For example,

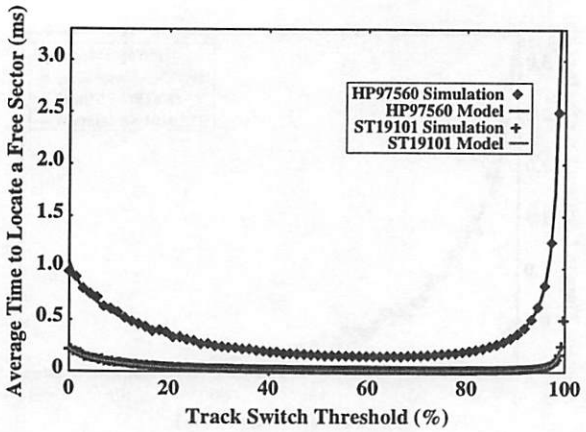


Figure 2: Average latency to locate free sectors for all writes performed to an initially empty track as a function of the track switch threshold, which is the percentage of free sectors reserved per track before a switch occurs. A high threshold corresponds to frequent switches.

the model simulates a single density zone while the actual disk has multiple zones. The simulated eager writing algorithm in Figure 1 is not restricted to the current cylinder and always seeks to the nearest sector. The figure shows that the single cylinder model is, in fact, a good approximation for an *entire* zone. This is because nearby cylinders are not much more likely than the current cylinder to have a free sector at an optimal rotational position and the head switch time is close to the single cylinder seek time.

Compared to the half-rotation delays of 7 ms for the HP and 3 ms for the Seagate that an update-in-place system may incur, Figure 1 promises significant performance improvement, especially at lower utilizations. Indeed, the figure shows that the latency has improved by nearly an order of magnitude on the newer Seagate disk compared to the HP disk. At higher disk utilizations, however, it takes longer to find a nearby free sector. One solution is to compact free space.

2.3 A Model Assuming a Compactor

Compacting free space using the disk processor can take advantage of the "free" bandwidth between the disk head and the platters during idle periods without consuming valuable I/O bus bandwidth, polluting host cache and memory, or interfering with host CPU operation. If we assume that we can compact free space during idle periods, then we need not worry about having to write to nearly full tracks. Instead, we can simply fill empty tracks to some threshold and rely on the compactor to generate more empty tracks. We will discuss the details of such compaction in Section 4.2; here we present the analysis of a system that makes such assumptions.

Suppose n is the total number of sectors in a track, m is the number of free sectors reserved per track before we switch tracks, s is the cost of a track switch, and r is the rotational delay incurred per sector, then the average latency incurred locating a free sector is:

$$\frac{s + r \cdot [(n+1)\ln\frac{n+2}{m+2} - (n-m) + \epsilon(n,m)]}{n-m} \quad (5)$$

where the ϵ function accounts for the non-randomness of the free space distribution. Appendix A.2 provides the derivation of the model.

Figure 2 validates the model of (5). When we switch tracks too frequently, although the free sectors in any particular track between switches are plentiful, we are penalized by the high switch cost. When we switch tracks too infrequently, locating free sectors becomes harder in a crowded track and the performance is also non-optimal. In general, the model aids the judicious selection of an optimal threshold for a particular set of disk parameters. It also reassures us that we need not suffer the performance degradation seen at high utilizations. Figure 1 and 2 also show why the demands on the compactor are less than the demands on the LFS cleaner. This is because the models indicate that compacting is only necessary for high utilizations and it can move data at small granularity.

3 A Virtual Log

Eager writing allows a high degree of location independence of data. It also presents two challenges: how to locate data as its physical location can constantly change, and how to recover this location information after a failure. In this section, we explain the rationale behind the design of the virtual log and the file systems built on it.

3.1 The Indirection Map

To support location independence, we introduce an *indirection map*, which maps logical addresses to physical disk addresses, a technique similar to those found in some existing systems [5, 7, 8]. In the rest of this section, we describe how we keep the map persistent, how we perform updates, how we recover the map after a failure, and how file systems can be built using these primitives. Our goal is to avoid some of the inefficiencies and inflexibilities of the previous approaches:

- scanning large portions of the disk for recovery,

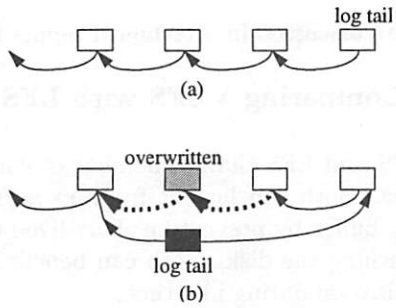


Figure 3: Maintaining the virtual log. (a) New map entry sectors are appended to a backward chain. (b) Implementing the backward chain as a tree to support map entry overwriting.

- reliance on NVRAM, which is not always available,
- excessive overhead in terms of space and extra I/O's needed to maintain the map, and
- altering the physical block format on disks to include, for example, a self-identifying header.

3.2 Maintaining and Recovering the Virtual Log

We implement the indirection map as a table. To keep the map persistent, we leverage the low latency offered by eager writing. Whenever an update takes place, we write the piece of the table that contains the new map entry to a free sector near the disk head. Suppose the file system addresses the disk at sector granularity and each map entry consumes four to eight bytes, the indirection map will consume a storage overhead between one to two percent of the disk capacity. We will discuss how to further reduce this storage overhead in the next section. If a transaction includes multiple data blocks and their map entries do not fall in the same map sector, then multiple map sectors may need to be written. Although the alternative of logging the multiple map entries using a single sector may better amortize the cost of map entry updates, it requires garbage collecting the obsolete map entries and is not used in our current design.

To recover the map after a failure, we must be able to identify the locations of the map sectors that are scattered throughout the disk due to eager writing. One way to accomplish this is to thread these sectors together to form a log. We term this a *virtual log* because its components are not necessarily physically contiguous. Because eager writing prevents us from predicting the location of the next map entry, we cannot maintain forward pointers in the map entries. Instead, we chain the map entries backward as shown in Figure 3a. Note that we can adapt this technique to a disk that supports a header per block, in which case a map entry including the backward pointer can be placed in the header.

As map entries are overwritten, the backward chain will accumulate obsolete sectors over time. We cannot simply reuse these obsolete sectors because doing so will break the backward chain. Our solution is to implement the backward linked list as a tree as shown in Figure 3b. Whenever an existing map entry is overwritten, a new log tail is introduced as the new tree root. One branch of the root points to the previous root; the other points to the map sector following the overwritten map entry. The overwritten sector can be recycled without breaking the virtual log. As Section 3.3 will show, we can keep the entire virtual log in disk memory during normal operation. Consequently, overwriting a map entry requires only one disk I/O to create the new log tail.

To recover the virtual log without scanning the disk, we must remember the location of the log tail. Modern disk drives use residual power to park their heads in a landing zone at the outer perimeter of the disks prior to spinning down the drives. It is easy to modify the firmware so that the drive records the current log tail location at a fixed location on disk before it parks the actuator[20, 31]. To be sure of the validity of the content stored at this fixed location, we can protect it with a checksum and clear it after recovery. In the extremely rare case when this power down sequence fails, we can detect the failure by computing the checksum and resort to scanning the disk for cryptographically signed map entries to retrieve the log tail.

With a stable log tail, recovering the virtual log is straightforward. We start at the log tail as the root and traverse the tree on the frontier based on age. Obsolete log entries can be recognized as such because their updated versions are younger and traversed earlier.

3.3 Implementing LFS on the Virtual Log

So far, we have described 1) a generic logging strategy that can support transactional behavior, and 2) an indirection map built on the log that can support location independence of data. One advantage of this approach is that we can implement eager writing behind a logical disk interface and deliver its performance advantage to an unmodified file system. We now describe another application of the virtual log: implementing a variant of the

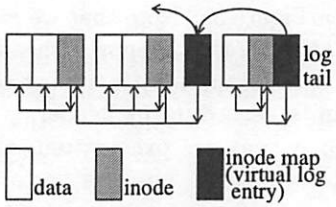


Figure 4: Implementing LFS on the virtual log. The virtual log contains only the inode map blocks.

log-structured file system (VLFS). Unlike the logical disk approach above, VLFS requires modifying the disk interface to the host file system. By seeking a tighter integration of the file system into the programmable disk, however, VLFS allows a number of optimizations impossible with an unmodified UFS. Currently, we have not directly implemented VLFS. Instead, the experiments in Section 5 are based on file systems running on the virtual log via the logical disk interface as described in the last section. We indirectly deduce the VLFS performance by evaluating these file systems.

One disadvantage of the indirection map as described in Section 3.1 is the amount of storage space and the extra I/O's needed to maintain and query the map. To solve this inefficiency, the file system can store physical addresses of the data blocks in the inodes, similar to the approach taken by LFS shown in Figure 4. As file blocks are written, the data blocks, the inode blocks that contain physical addresses of the data blocks, and inode maps that contain physical addresses of the inodes are all appended to the log. What is different in the virtual log based implementation (VLFS) is that the log need not be physically contiguous, and only the inode map blocks logically belong to the virtual log. This is essentially adding a level of indirection to the indirection map. The advantage is that the inode map, which is the sole content of the virtual log, is now compact enough to be stored in memory; it also reduces the number of I/O's needed to maintain the indirection map because VLFS simply takes advantage of the existing indirection data structures in the file system without introducing its own.

Another LFS optimization that can also be applied to VLFS is checkpointing for recovery. Periodically, we write the entire inode map to the disk contiguously. At recovery time, while LFS reads a checkpoint at a known disk location and rolls forward, VLFS traverses the virtual log backwards from the log tail towards the checkpoint.

VLFS also opens up a variety of questions including how to re-engineer the host/disk interface and how to implement the free space compactor. These issues are discussed in a technical report [35].

3.4 Comparing VLFS with LFS

VLFS and LFS share a number of common advantages. Both can benefit from an asynchronous memory buffer by preventing short-lived data from ever reaching the disk. Both can benefit from disk reorganization during idle time.

Due to eager writing, VLFS possesses a number of unique advantages. First, small synchronous writes perform well on VLFS whereas the LFS performance suffers if an application requires frequent "fsync" operations. Second, while the free space compactor is only an optimization for VLFS, the cleaner is a necessity for LFS. In cases where idle time is scarce or disk utilization is high, VLFS can avoid the bandwidth waste incurred during repeated copying of live data by the LFS cleaner [22, 29, 30]. Third, LFS needs large idle intervals to mask the cleaning overhead because it moves data at large segment granularity. The VLFS compactor, however, can take advantage of short idle intervals. Finally, reads can interfere with LFS writes by forcing the disk head away from free space and/or disturbing the track buffer (which can be sometimes used to absorb writes without accessing the disk platters). VLFS can efficiently perform intervening writes near the data being read.

VLFS and LFS also share some common disadvantages. For example, data written randomly may have poor sequential read performance. In some of these situations, reorganization techniques that can improve LFS performance [22] should be equally applicable to VLFS. In some other situations, aggressive prefetching [24] and "disk-directed I/O" [16] can also serve the virtual log well.

4 Experimental Platform

We evaluate the following four combinations of file systems and simulated disks (shown in Figure 5): a UFS on a regular disk, a UFS on a Virtual Log Disk (VLD), an LFS on a regular disk, and an LFS on a VLD. Although we do recognize that a complete evaluation of the VLFS would require a complete implementation of VLFS, in this paper, we take the first step of deducing the behavior of VLFS by examining the file systems running on the VLD.

Two host machines are used: one is a 50 Mhz SUN SPARCstation-10, which is equipped with 64 MB of memory and runs Solaris 2.6; the other is

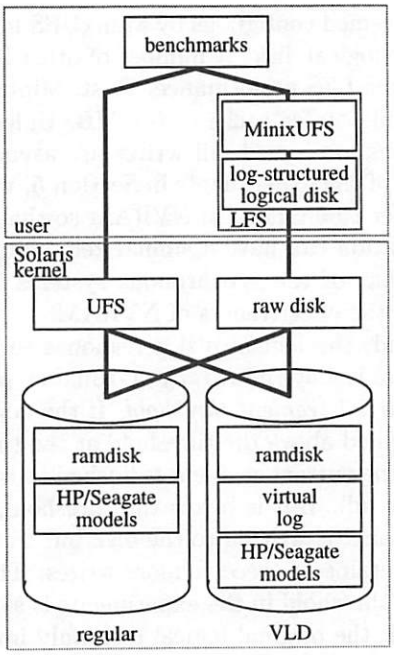


Figure 5: Architecture of the experimental platform.

a similarly configured UltraSPARC-170 workstation that runs at 167 Mhz. The SPARCstation-10 supports both 4 KB and 8 KB file blocks while the UltraSPARC-170 only supports 8 KB file blocks. Because our focus is small write performance, we run our experiments on the SPARCstation-10 unless explicitly stated to be otherwise. We perform some additional experiments on the UltraSPARC-170 only to study the impact of host processing speed. We next briefly describe each of the disk modules and file systems shown in Figure 5.

4.1 The Regular Disk

The regular disk module simulates a portion of the HP97560 disk or the Seagate ST19101 disk, whose relevant parameters are shown in Table 1. A ramdisk driver is used to store file data using 24 MB of kernel memory. The Dartmouth simulator [17] is ported into the kernel to ensure realistic timing behavior of the HP disk. We adjust the parameters of the Dartmouth model to coincide with those of the Seagate disk to simulate the faster disk. Although not as accurate, the Seagate model gives a clear indication of the impact of disk technology improvements. We have run all our experiments on both disk models. Unless stated otherwise, however, we present the results obtained on the Seagate model.

The ramdisk simulator allows two simulation modes. In one mode, the simulator sleeps the right amount of time reported by the Dartmouth model and we can conduct the evaluations by directly timing the application. In the second mode, the simulator runs at memory speed without sleeping. This mode speeds up certain phases of the experiments whose actual elapsed time is not important. The disadvantage of the ramdisk simulator is its small size due to the limited kernel memory. We only simulate 36 cylinders of the HP97560 and 11 cylinders of the Seagate.

4.2 The Virtual Log Disk

The VLD adds the virtual log to the disk simulator described above. It exports the same device driver interface so it can be used by existing file systems. The VLD maintains an in-memory indirection map while updating the on-disk representation as described in Section 3.2. To avoid trapping the disk head in regions of high utilization during eager writing, the VLD performs cylinder seeks only in one direction until it reaches the last cylinder, at which point it starts from the first cylinder again.

One challenge of implementing the VLD while preserving the existing disk interface is handling deletes, which are not visible to the device driver. This is a common problem faced by logical disks. Our solution is to monitor overwrites: when a logical address is re-used, the VLD detects that the old logical-to-physical mapping can be freed. The disadvantage of the approach is that it does not capture the freed blocks that are not yet overwritten.

Another question that arose during the implementation of the VLD is the choice of the physical disk block size. As shown in Appendix A.1, the latency is best when the physical block size matches the file system logical block size. In all our experiments, we have used a physical block size of 4 KB. The resulting internal fragmentation when writing data or metadata blocks that are smaller only biases against the performance of UFS running on the VLD. Each physical block requires a four byte map entry; so the entire map consumes 24 KB.

The third issue concerns the interaction between eager writing and the disk track buffer read-ahead algorithm. When reading, the Dartmouth simulator keeps in cache only the sectors from the beginning of the current request through the current read-ahead point and discards the data whose addresses are lower than that of the current request. This algorithm makes sense for sequential reads of data whose physical addresses increase monotonically. This is the case for traditional disk allocation strategies. For VLD, however, the combination of eager writing and the logical-to-physical address

translation means that the sequentially read data may not necessarily have monotonically increasing physical addresses. As a result, the Dartmouth simulator tends to purge data prematurely from its read-ahead buffer under VLD. The solution is to aggressively prefetch the entire track as soon as the head reaches the target track and not discard data until it is delivered to the host during sequential reads. The measurements of sequential reads on the VLD in Section 5 were taken with this modification.

Lastly, the VLD module also implements a free space compactor. Although the eager writing strategy should allow the compactor to take advantage of idle intervals of arbitrary length, for simplicity, our compactor compacts free space at the granularity of tracks. During idle periods, the compactor reads the current track and uses eager writing to copy the live data to other tracks in a way similar to hole-plugging under LFS [22, 36]. Currently, we choose compaction targets randomly and plan to investigate optimal VLD compaction algorithms (e.g., ones that preserve or enhance read and write locality) in the future. Applying the lessons learned from the models in Section 2.3, the VLD fills empty tracks to a certain threshold (75% in the experiments). After exhausting empty tracks generated by the compactor, the VLD reverts to the greedy algorithm modeled in Section 2.2.

4.3 UFS

Because both the regular disk and the VLD export the standard device driver interface, we can run the Solaris UFS (subsequently also labeled as UFS) unmodified on these disks. We configure UFS with a block size of 4 KB and a fragment size of 1 KB. Like most other Unix file systems, the Solaris UFS updates metadata synchronously while the user can specify whether the data writes are synchronous. It also performs prefetching after several sequential reads are detected.

4.4 LFS

We have ported the MIT Log-Structured Logical Disk (LLD) [7], a user level implementation of LFS (subsequently also labeled as LFS). It consists of two modules: the MinixUFS and the log-structured logical disk, both running at user level. MinixUFS accesses the logical disk with a block interface while the logical disk interfaces with the raw disk using segments. The block size is 4 KB and the segment size is 0.5 MB. The implementors of LLD has disabled read-ahead in MinixUFS because blocks deemed contiguous by MinixUFS may not be so in the logical disk. A number of other issues also impact the LFS performance. First, MinixUFS employs a file buffer cache of 6.1 MB. Unless "sync" operations are issued, all writes are asynchronous. In some of the experiments in Section 5, we assume this buffer to be made of NVRAM so that the LFS configuration can have a similar reliability guarantee as that of the synchronous systems. We will examine the effectiveness of NVRAM.

Second, the logical disk's response to a "sync" operation is determined by a tunable parameter called *partial segment threshold*. If the current segment is filled above the threshold at the time of the "sync", the current segment is flushed to the disk as if it were full. If it is below the threshold, the current segment is written to the disk but the memory copy is retained to receive more writes. The partial segment threshold in the experiments is set to 75%.

Third, the original logical disk only invokes the cleaner when it runs out of empty segments. We have modified the cleaner so that it can be invoked during idle periods before it runs out of free space.

5 Experimental Results

In this section, we compare the performance of eager writing against that of update-in-place and logging with a variety of micro-benchmarks. We first run the small file and large file benchmarks that are commonly used by similar file system studies. Then we use a benchmark that best demonstrates the strength of eager writing: small random synchronous updates with no idle time, which also illustrates the effect of disk utilization. Next we examine the effect of technology trends. Last, we examine how the availability of idle time impacts the performance of eager writing and logging. Unless explicitly stated to be otherwise, the experimental platform is the SPARCstation-10 running on the simulated Seagate disk.

5.1 Small File Performance

We first examine two benchmarks similar to the ones used by both the original LFS study [25] and the Logical Disk study [7]. In the first benchmark, we create 1500 1 KB files, read them back after a cache flush, and delete them. The benchmark is run on empty disks. The results of this benchmark are shown in Figure 6. Under LFS, updates are flushed to disk only if the memory buffer is filled. Under UFS, updates are synchronous. Due to the different

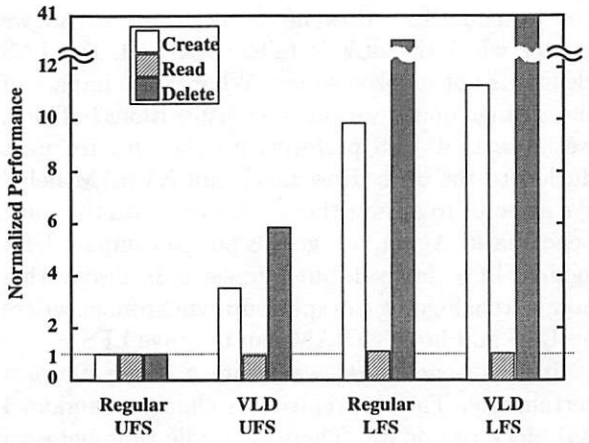


Figure 6: Small file performance. The benchmark creates, reads, and deletes 1 KB files. All performance is normalized to that of UFS running on a regular disk.

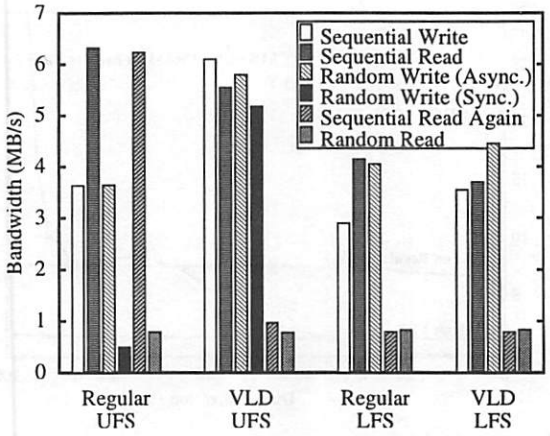


Figure 7: Large file performance. The benchmark sequentially writes a 10 MB file, reads it back sequentially, writes it again randomly (both asynchronously and synchronously for the UFS runs), reads it again sequentially, and finally reads it randomly.

reliability guarantees of UFS and LFS, this experiment is not designed to compare UFS against LFS. Instead, our interest is in examining the impact of virtual logging on both file systems.

As expected, VLD significantly speeds up UFS during the create and delete phases due to eager writing's ability to complete small writes more quickly than update-in-place. The read performance on the VLD is slightly worse than that on the regular disk because of the overhead introduced by the indirection map and the fact that disk read-ahead is not as effective. The same pattern in read performance recurs in other experiments as well.

VLD also speeds up LFS writes slightly. The reasons are the occasional miss of rotations and long-distance seeks between segment-sized writes on the regular disk, which the VLD diligently avoids.

Ignoring the cost of LFS cleaning (which is not triggered in this experiment), we speculate that the impact of VLD on a UFS that employs delayed write techniques (such as those proposed by Ganger [9]) should be between that of the unmodified UFS and that of LFS. Like LFS, however, delayed writes under UFS do not guarantee data reliability.

From this benchmark, we speculate that by integrating LFS with the virtual log, the VLFS (which we have not implemented) should approximate the performance of UFS on the VLD when we must write synchronously, while retaining the benefits of LFS when asynchronous buffering is acceptable.

5.2 Large File Performance

In the second benchmark, we write a 10 MB file sequentially, read it back sequentially, write 10 MB of data randomly to the same file, read it back sequentially again, and finally read 10 MB of random data from the file. The performance of random I/O can also be an indication of the effect of interleaving a large number of independent concurrent workloads. The benchmark is again run on empty disks. Figure 7 shows the results. The writes are asynchronous with the exception of the two random write runs on UFS that are labeled as "Sync". Neither the LFS cleaner nor the VLD compactor is run.

We first point out a few characteristics that are results of implementation artifacts. The first two phases of the LFS performance are not as good as those of UFS because the user level LFS implementation is less efficient than the native in-kernel UFS. LFS also disables prefetching, which explains its low sequential read bandwidth. Sequential reads on UFS run much faster than sequential writes on the regular disk due to aggressive prefetching both at the file system level and inside the disk. With these artifacts explained, we now examine a number of interesting performance characteristics.

First, sequential read after random write performs poorly in all LFS and VLD systems because both logging and eager writing destroy spatial locality. This is a problem that may be solved by a combination of caching, data reorganization, hints at interfaces, and prefetching as explained in Section 3.4.

Second, the LFS random write bandwidth is higher than that of sequential write. This is because during the random write phase, some of the blocks are written multiple times and fewer bytes reach the disk. This is a benefit of delayed writes.

Third, on a UFS, while it is not surprising that

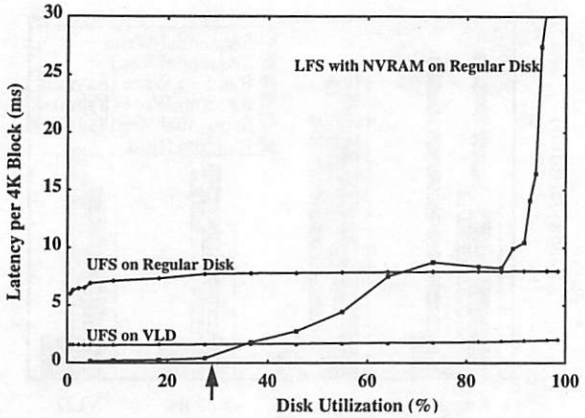


Figure 8: Performance of random small synchronous updates under various disk utilizations. The disk utilization is obtained from the Unix "df" utility and includes about 12% of reserved free space that is not usable. The arrow on the x-axis points to the size of the NVRAM used by LFS.

the synchronous random writes do well on the VLD, it is interesting to note that even sequential writes perform better on the VLD. This is because of the occasional inadvertent miss of disk rotations on the regular disk. Interestingly enough, this phenomenon does not occur on the slower HP97560 disk. This evidence supports our earlier contention that tuning the host operating system to match changing technologies is indeed a difficult task. The approach of running the file system inside the disk in general and the concept of a virtual log in particular can simplify such efforts.

Fourth, although our regular disk simulator does not implement disk queue sorting, UFS does sort the asynchronous random writes when flushing to disk. Therefore, the performance of this phase of the benchmark, which is also worse on the regular disk than on the VLD due to the reason described above, is a best case scenario of what disk queue sorting can accomplish. In general, disk queue sorting is likely to be even less effective when the disk queue length is short compared to the working set size. Similar phenomenon can happen for a write-ahead logging system whose log is small compared to the size of the database. The VLD based systems need not suffer from these limitations. In summary, the benchmark further demonstrates the power of combining lazy writing by the file system with eager writing by the disk.

5.3 Effect of Disk Utilization

There are a number of questions that are still unanswered by the first two benchmarks. First, the VLD always has plenty of free space in the previous benchmarks. How much degradation can we expect when the disk is fuller? Second, the LFS cleaner is not invoked so far. What is the impact of the cleaner under various disk utilizations? Third, we know that LFS performs poorly with frequent flushes to the disk. How much can NVRAM help? We attempt to answer these questions with the third benchmark. Again, our goal is not to compare UFS against LFS. Instead, our interest is in discovering how virtual logging can speed up synchronous writes on UFS and how NVRAM can improve LFS.

In this benchmark, we create a single file of a certain size. Then we repeatedly choose a random 4 KB block to update. There is no idle time between writes. For UFS, the "write" system call does not return until the block is written to the disk surface. For LFS, we assume that the 6.1 MB file buffer cache is made of NVRAM and we do not flush to disk until the buffer cache is full. We measure the steady state bandwidth of UFS on the regular disk, UFS on the VLD, and LFS on the regular disk as we vary the size of the file we update.

Figure 8 plots the average latency experienced per write. UFS on the regular disk suffers from excessive disk head movement due to the update-in-place policy. The latency increases slightly as the updated file grows because the disk head needs to travel a greater distance between successive writes. This increase may have been larger had we simulated the entire disk.

LFS provides excellent performance when the entire file fits in NVRAM. As soon as the file outgrows the NVRAM, writes are forced to the disk; as this happens and as disk utilization increases, the cleaner quickly dominates performance. The plateau between roughly 60% and 85% disk utilization is due to the fact that the LFS cleaner chooses less utilized segments to clean; with certain distribution patterns of free space, the number of segments to clean in order to generate the same amount of free segments may be the same as (or even larger than) that that required under a higher utilization.

With eager writing, the VLD suffers from neither the excessive disk head movements, nor the bandwidth waste during cleaning. As the disk utilization increases, the VLD latency also rises. The rise, however, is not significant compared to the various overheads in the system, which we examine next.

5.4 Effect of Technology Trends

The performance improvement provided by eager writing seen so far is not as great as the analytical models might suggest. To see why this is the case,

HP SPARC	Seagate SPARC	Seagate UltraSPARC
2.6×	5.1×	9.9×

Table 2: Performance gap between update-in-place and virtual-logging widens as disks and processors improve.

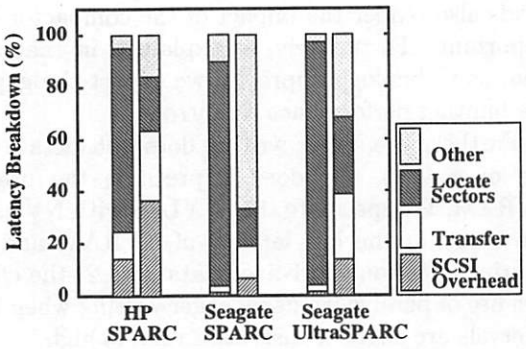


Figure 9: Breaking down the total latency into SCSI overhead, transfer time, time required to locate free sectors, and other processing time. Update-in-place performance (left bars) becomes increasingly dominated by mechanical delays while virtual logging (right bars) achieves a balance between processor and disk improvements.

we now provide a more detailed breakdown of the benchmark times reported in the last section. We also examine the impact of technology trends.

We repeat the UFS experiment of the last section on three different platforms under the same disk utilization (80%)[1]. Table 2 shows the result. The first column shows the speed-up of virtual logging compared to update-in-place on a SPARCstation-10 and HP disk combination. Next, we replace the older HP disk with the new Seagate disk. In the third run, we replace the older SPARCstation-10 with the newer UltraSPARC-170. We see that the performance gap widens to almost an order of magnitude.

Figure 9 reveals the underlying source of this performance difference by providing the detailed breakdown of the latencies. The "SCSI overhead" component is the time that the disk processor spends processing each SCSI command. The "transfer" component is the time it takes to move the bits to or from the media after the head has been positioned over the target sector. The component labeled as "locate sectors" is the amount of time the disk spends positioning the disk head. It includes seek, rotation, and head switch times. The "other" component includes the operating system processing overhead, which includes the time to run the virtual log algorithm because the disk simulator is

[1]The VLD latency in this case is measured immediately after running a compactor, as explained in Section 5.5.

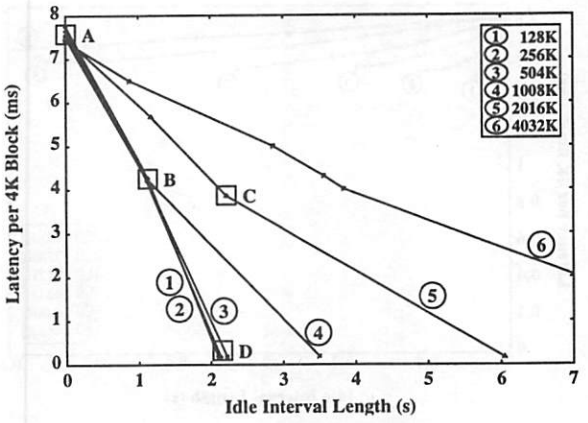


Figure 10: Performance of LFS (with NVRAM) as a function of available idle time.

part of the host kernel. The time consumed by simulating the disk mechanism itself is less than 5% of this component.

We see that the mechanical delay becomes a dominant factor of update-in-place latency. We also see that eager writing has indeed succeeded in significantly reducing the disk head positioning times. The overall performance improvement on the older disk or on the older host, however, is low due to the high overheads. After we replace the older disk, the performance of virtual logging becomes host limited as the "other" component dominates. After we replace the older host, however, the latency components again become more balanced. This indicates that the virtual log is able to ride the impressive disk bandwidth growth, achieving a balance between processor and disk improvements.

5.5 Effect of Available Idle time

The benchmark in Section 5.3 assumes zero idle time. This severely stresses LFS because the cleaning time is not masked by idle periods. It also penalizes the VLD by disallowing free space compacting. In this section, we examine how LFS on a regular disk and how UFS on a VLD may benefit from idle time. We modify the benchmark of Section 5.3 to perform a burst of random updates, pause, and repeat. The disk utilization is kept at 80%.

Figure 10 shows how the LFS performance responds to the increase of idle interval length. Each curve represents a different burst size. At point A, no idle time is available. LFS fills up the NVRAM with updates and flushes the file buffer to the regular disk, invoking the cleaner when necessary.

At point B, enough idle time is available to clean one segment. If the burst size is less than or equal

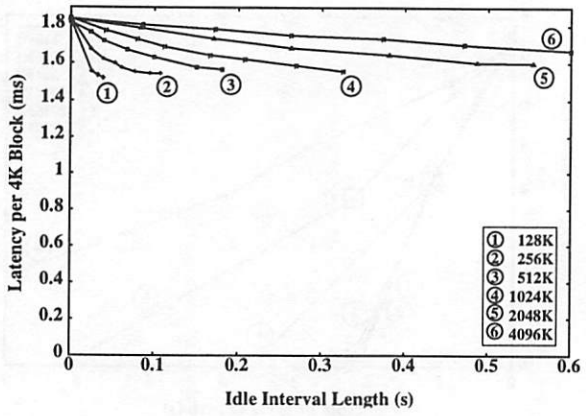


Figure 11: Performance of UFS on VLD as a function of available idle time.

to 1 MB, at the time when the NVRAM becomes full, the cleaner has already generated the maximum number of empty segments possible. Consequently, flushing the NVRAM takes a constant amount of time[2]. A similar latency is achieved at point C where two segments are cleaned per idle interval.

Point D corresponds to sufficient idle time to flush the entire burst from NVRAM to disk. The benchmark runs at memory speed because the cleaning time during flushing is entirely masked by the long idle periods. The first three curves coincide at this point because these burst sizes fit in a single segment.

We do not pretend to know the optimal LFS cleaning algorithm using NVRAM, which is outside the scope of this paper. Nevertheless, two facts are apparent. First, because the cleaner moves segment-sized data, LFS can only benefit from relatively long idle intervals. Second, unless there is sufficient idle time to mask the flushing of the burst buffered in the NVRAM, LFS performance remains poor.

Figure 11 shows the result of repeating the same benchmark on UFS running on a VLD. Unlike the LFS cleaner, the VLD compactor has no restriction in terms of the granularity of the data it moves. For convenience, our implementation of the compactor moves data at track granularity, which is still much smaller than the LFS segment granularity. The performance on the VLD improves along a continuum of relatively small idle intervals. The VLD performance is also more predictable, whereas LFS performance experiences large variances depending on whether the NVRAM is full and whether cleaning is necessary. The experiments of this section run on the SPARCstation-10. As we have seen in the last section, the more powerful UltraSPARC-170 can easily cut the latency in Figure 11 in half.

The disadvantage of UFS on the VLD compared to LFS with NVRAM is the limiting performance with infinite amount of idle time. Because each write reaches the disk surface, the VLD experiences the overheads detailed in Section 5.4. The overheads also render the impact of the compactor less important. Fortunately, as explained in that section, as technology improves, we expect the gap of the limiting performance to narrow.

Furthermore, eager writing does not dictate the use of a UFS, nor does it preclude the use of NVRAM. We speculate that a VLFS with NVRAM can enjoy 1) the low latency of NVRAM and its filtering effect on short-lived data, and 2) the effective use of bandwidth using eager writing when idle intervals are short or disk utilization is high.

6 Related Work

The work presented in this paper builds upon a number of existing techniques including reducing latency by writing data near the disk head, a transactional log, file systems that support data location independence, and log-structured file systems. The goal of this study is not to demonstrate the effectiveness of any of these individual ideas. Rather, the goals are 1) provide a theoretical foundation of eager writing with the analytical models, 2) show that the integration of these ideas at the disk level can provide a number of unique benefits to both UFS and LFS, 3) demonstrate the benefits of eager writing without the semantic compromise of delayed writes or extra hardware support such as NVRAM, and 4) conduct a series of systematic experiments to quantify the differences of the alternatives. We are not aware of existing studies that aim for these goals.

Simply writing data near the disk head is not a new idea. Many efforts have focused on improving the performance of the write-ahead log. This is motivated by the observation that appending to the log may incur extra rotational delay even when no seek is required. The IBM IMS Write Ahead Data Set (WADS) system [10] addresses this issue for drums (fixed-head disks) by keeping some tracks completely empty. Once each track is filled with a single block, it is not re-used until the data is copied out of the track into its normal location.

Likewise, Hagmann places the write-ahead log in its own logging disk[13], where each log append can fill any open block in the cylinder until its utilization

[2] In this case, because the NVRAM size is larger than the available free space, the cleaner still needs to run during flushing to reclaim the free space created during flushing.

reaches a threshold. Eager writing in our system, while retaining good logging performance, assumes no dedicated logging disk and does not require copying of data from the log into its permanent location. The virtual log *is* the file system.

A number of disk systems have also explored the idea of lowering latency of small writes by writing near the disk head location. Menon proposes to use this technique to speed up parity updates in disk arrays [23]. Under *parity logging*, an update is performed to a rotationally optimal position in a cylinder. It relies on NVRAM to keep the indirection map persistent.

Mime [5], the extension of Loge [8], also writes near disk head and is the closest in spirit to our system. There are a number of differences between Mime and the virtual log. First, Mime relies on self-identifying disk blocks. Second, Mime scans free segments to recover its indirection map. As disk capacity increases, this scanning may become a time consuming process. Third, the virtual log also incorporates a free space compactor.

The Network Appliance file system, WAFL [14, 15], checkpoints the disk to a consistent state periodically, uses NVRAM for fast writes between checkpoints, and can write data and metadata anywhere on the disk. An exception of the write-anywhere policy is the root inodes, which are written for each checkpoint and must be at fixed locations. Unlike Mime, WAFL supports fast recovery by rolling forward from a checkpoint using the log in the NVRAM. One goal of the virtual log is to support fast transactions and recovery without NVRAM, which has capacity, reliability, and cost limitations. Another difference is that the WAFL write allocation decisions are made at the RAID controller level, so the opportunity to optimize for rotational delay is limited.

Our idea of fast atomic writes using the virtual log originated as a generalization of the AutoRAID technique of hole-plugging [36] to improve LFS performance at high disk utilizations without AutoRAID hardware support for self-describing disk sectors. In hole-plugging, partially empty segments are freed by writing their live blocks into the holes found in other segments. This outperforms traditional cleaning at high disk utilizations by avoiding reading and writing a large number of nearly full segments [22]. AutoRAID requires an initial log-structured write of a physically contiguous segment, after which it is free to copy the live data in a segment into any empty block on the disk. In contrast, the virtual log eliminates the initial segment write and can efficiently schedule the individual writes.

7 Conclusion

In this paper, we have developed a number of analytical models that show the theoretical performance potential of eager writing, the technique of performing small atomic writes near the disk head position. We have presented the virtual log design which supports fast synchronous writes without the semantic compromise of delayed writes or extra hardware support such as NVRAM. This design requires careful log management for fast recovery. By conducting a systematic comparison of this approach against traditional update-in-place and logging approaches, we have demonstrated the benefits of applying the virtual log approach to both UFS and LFS. The availability of fast synchronous writes can also simplify the design of file systems and sophisticated applications. As the current technology trends continue, we expect that the performance advantage of this approach will become increasingly important.

Acknowledgements

We would like to thank Arvind Krishnamurthy for many interesting discussions on the proofs of several analytical models, Marvin Solomn for discovering the simplest proof of the single track model, Daniel Stodolsky and Chris Malakapalli for helping us understand the working of Quantum and Seagate disks, Jeanna Neefe Matthews for asking the question of how hole-plugging could be efficiently implemented without hardware support for self-describing disk sectors, Geoff Voelker and John Wilkes for comments on early drafts and presentations, and Margo Seltzer for a large number of excellent suggestions during the shepherding process.

References

[1] ADAMS, L., AND OU, M. Processor Integration in a Disk Controller. *IEEE Micro 17*, 4 (July 1997).

[2] ATKINSON, M., CHISHOLM, K., COCKSHOTT, P., AND MARSHALL, R. Algorithms for a Persistent Heap. *Software - Practice and Experience 13*, 3 (March 1983), 259–271.

[3] BAKER, M., ASAMI, S., DEPRIT, E., OUSTERHOUT, J., AND SELTZER, M. Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings of the*

Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V) (Sept. 1992), pp. 10–22.

[4] BIRRELL, A., HISGEN, A., JERIAN, C., MANN, T., AND SWART, G. The Echo Distributed File System. Technical Report 111, Digital Equipment Corp. Systems Research Center, Sept. 1993.

[5] CHAO, C., ENGLISH, R., JACOBSON, D., STEPANOV, A., AND WILKES, J. Mime: a High Performance Parallel Storage Device with Strong Recovery Guarantees. Tech. Rep. HPL-CSP-92-9 rev 1, Hewlett-Packard Company, Palo Alto, CA, March 1992.

[6] CHUTANI, S., ANDERSON, O., KAZAR, M., LEVERETT, B., MASON, W., AND SIEDBOTHAM, R. The Episode File System. In *Proc. of the 1992 Winter USENIX* (January 1992), pp. 43–60.

[7] DE JONGE, W., KAASHOEK, M. F., AND HSIEH, W. C. The Logical Disk: A New Approach to Improving File Systems. In *Proc. of the 14th ACM Symposium on Operating Systems Principles* (December 1993), pp. 15–28.

[8] ENGLISH, R. M., AND STEPANOV, A. A. Loge: a Self-Organizing Disk Controller. In *Proc. of the 1992 Winter USENIX* (January 1992).

[9] GANGER, G. R., AND PATT, Y. N. Metadata Update Performance in File Systems. In *Proc. of the First Symposium on Operating Systems Design and Implementation* (November 1994), pp. 49–60.

[10] GAWLICK, D., GRAY, J., LIMURA, W., AND OBERMARCK, R. Method and Apparatus for Logging Journal Data Using a Log Write Ahead Data Set. U.S. Patent 4507751 issued to IBM, March 1985.

[11] GROCHOWSKI, E. G., AND HOYT, R. F. Future Trends in Hard Disk Drives. *IEEE Transactions on Magnetics 32*, 3 (May 1996).

[12] HAGMANN, R. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proc. of the 11th ACM Symposium on Operating Systems Principles* (October 1987), pp. 155–162.

[13] HAGMANN, R. Low Latency Logging. Tech. Rep. CSL-91-1, Xerox Corporation, Palo Alto, CA, February 1991.

[14] HITZ, D., LAU, J., AND MALCOLM, M. File System Design for an NFS File Server Appliance. In *Proc. of the 1994 Winter USENIX* (January 1994).

[15] HITZ, D., LAU, J., AND MALCOLM, M. File System Design for an NFS File Server Appliance. Tech. Rep. 3002, Network Appliance, March 1995.

[16] KOTZ, D. Disk-directed I/O for MIMD Multiprocessors. In *Proc. of the First Symposium on Operating Systems Design and Implementation* (November 1994), pp. 61–74.

[17] KOTZ, D., TOH, S., AND RADHAKRISHNAN, S. A Detailed Simulation Model of the HP 97560 Disk Drive. Tech. Rep. PCS-TR91-220, Dept. of Computer Science, Dartmouth College, July 1994.

[18] LAMB, C., LANDIS, G., ORENSTEIN, J., AND WEINREB, D. The ObjectStore Database System. *Communications of the ACM 34*, 10 (October 1991), 50–63.

[19] LOWELL, D. E., AND CHEN, P. M. Free Transactions with Rio Vista. In *Proc. of the 16th ACM Symposium on Operating Systems Principles* (October 1997), pp. 92–101.

[20] MALAKAPALLI, C. Personal Communication, Seagate Technology, Inc., July 1998.

[21] MASHEY, J. R. Big Data and the Next Wave of InfraStress. Computer Science Division Seminar, University of California, Berkeley, October 1997.

[22] MATTHEWS, J. N., ROSELLI, D. S., COSTELLO, A. M., WANG, R. Y., AND ANDERSON, T. E. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proc. of the 16th ACM Symposium on Operating Systems Principles* (October 1997), pp. 238–251.

[23] MENON, J., ROCHE, J., AND KASSON, J. Floating parity and data disk arrays. *Journal of Parallel and Distributed Computing 17*, 1 and 2 (January/February 1993), 129–139.

[24] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed Prefetching and Caching. In *Proceedings of the ACM Fifteenth Symposium on Operating Systems Principles* (December 1995).

[25] ROSENBLUM, M., AND OUSTERHOUT, J. The Design and Implementation of a Log-Structured File System. In *Proc. of the 13th Symposium on Operating Systems Principles* (Oct. 1991), pp. 1–15.

[26] RUEMMLER, C., AND WILKES, J. An Introduction to Disk Drive Modeling. *IEEE Computer 27*, 3 (March 1994), 17–28.

[27] SATYANARAYANAN, M., MASHBURN, H. H., KUMAR, P., STEERE, D. C., AND KISTLER, J. J. Lightweight Recoverable Virtual Memory. In *Proc. of the 14th ACM Symposium on Operating Systems Principles* (December 1993), pp. 146–160.

[28] SEAGATE TECHNOLOGY, INC. Cheetah Specifications. http://204.160.183.162/disc/cheetah/cheetah.shtml, 1998.

[29] SELTZER, M., BOSTIC, K., MCKUSICK, M., AND STAELIN, C. An Implementation of a Log-Structured File System for UNIX. In *Proc. of the 1993 Winter USENIX* (Jan. 1993), pp. 307–326.

[30] SELTZER, M., SMITH, K., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File System Logging Versus Clustering: A Performance Comparison. In *Proc. of the 1995 Winter USENIX* (Jan. 1995).

[31] STODOLSKY, D. Personal Communication, Quantum Corp., July 1998.

[32] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS File System. In *Proc. of the 1996 Winter USENIX* (January 1996), pp. 1–14.

[33] TRANSACTION PROCESSING PERFORMANCE COUNCIL. *TPC Benchmark B Standard Specification.* Waterside Associates, Fremont, CA, Aug. 1990.

[34] TRANSACTION PROCESSING PERFORMANCE COUNCIL. *TPC Benchmark C Standard Specification.* Waterside Associates, Fremont, CA, August 1996.

[35] WANG, R. Y., ANDERSON, T. E., AND PATTERSON, D. A. Virtual Log Based File Systems for a Programmable Disk. Tech. Rep. UCB/CSD 98/1031, University of California at Berkeley, December 1998.

[36] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID Hierarchical Storage System. In *Proc. of the 15th ACM Symposium on Operating Systems Principles* (December 1995), pp. 96–108.

A.1 Proof and Extension of the Single Track Model

Suppose a disk track contains n sectors, its free space percentage is p, and the free space is randomly distributed. Section 2.1 gives the average number of sectors the disk head must skip before arriving at a free sector:

$$\frac{(1-p)n}{1+pn} \tag{6}$$

Proof. Suppose k is the number of free sectors in the track and $E(n,k)$ is the expected number of used sectors we encounter before reaching the first free sector. With a probability of k/n, the first sector is free and the number of sectors the disk head must skip is 0. Otherwise, with a probability of $(n-k)/n$, the first sector is used and we must continue searching in the remaining $(n-1)$ sectors, of which k are free. Therefore, the expected delay in this case is $[1+E(n-1,k)]$. This yields the recurrence:

$$E(n,k) = \frac{n-k}{n}[1+E(n-1,k)] \tag{7}$$

By induction on n, it is easy to prove that the following is the unique solution to (7):

$$E(n,k) = \frac{n-k}{1+k} \tag{8}$$

(6) follows if we substitute k in (8) with pn. □

Formula (6) models the latency to locate a single sector (or a block). To extend it to situations where we need to find multiple disk sectors (or blocks) to satisfy a single file system block allocation, consider the two following examples. First, let us suppose that the host file system writes 4 KB blocks. If the disk allocates and frees 512 B sectors, it may need to locate eight separate free sectors to complete eager writing one logical block. If the disk uses a physical block size of 4 KB instead, although it takes longer to locate the first free 4 KB-aligned sector, it is guaranteed to have eight consecutive free sectors afterwards. In general, suppose the file system logical block size is B and the disk physical block size is b ($b \leq B$), then the average amount of time (expressed in the numbers of sectors skipped) needed to locate all the free sectors for a logical block is:

$$\frac{(1-p)n}{b+pn} \cdot B \tag{9}$$

Formula (9) indicates that the latency is lowest when the physical block size matches the logical block size.

A.2 Derivation of the Model Assuming a Compactor

We derive the model of Section 2.3. If writes arrive with no delay, then it would be trivial to fill the track. If writes arrive randomly and we assume the free space distribution is random at the time of the arrivals, then writes between successive track switches follow the model of (6). Therefore, substituting pn in (6) with i, the number of free sectors, the total number of sectors to skip between track switches can be expressed as:

$$\sum_{i=m+1}^{n} \frac{n-i}{1+i} \tag{10}$$

where n is the total number of sectors in a track and m is the number of free sectors reserved per track before switching tracks. Suppose each track switch costs s and the rotation delay of one sector is r; because we pay one track switch cost per $(n-m)$ writes, the average latency per sector of this strategy can be expressed as:

$$\frac{s + r \cdot \sum_{i=m+1}^{n} \frac{n-i}{1+i}}{n-m} \tag{11}$$

So far, we have assumed that the free space distribution is always random. This is not the case with the approach of filling the track up to a threshold because, for example, the first free sector immediately following a number of used sectors is more likely to be selected for eager writing than one following a number of free sectors. In general, this non-randomness increases latency. Although the precise modeling of the non-randomness is quite complex, through approximations and empirical trials, we have found that adding the following ϵ function to (10) works well for a wide range of disk parameters in practice:

$$\epsilon(n,m) = \frac{(n-m-0.5)^{p+2}}{(8-\frac{n}{96}) \cdot (p+2) \cdot n^p} \tag{12}$$

where $p = 1 + n/36$. By approximating the summation in (10) with an integral and adding the correction factor of (12) to account for the non-randomness, we arrive at the final latency model:

$$\frac{s + r \cdot [(n+1)\ln\frac{n+2}{m+2} - (n-m) + \epsilon(n,m)]}{n-m} \tag{13}$$

Resource containers: A new facility for resource management in server systems

Gaurav Banga Peter Druschel
Dept. of Computer Science
Rice University
Houston, TX 77005
{gaurav,druschel}@cs.rice.edu

Jeffrey C. Mogul
Western Research Laboratory
Compaq Computer Corporation
Palo Alto, CA 94301
mogul@pa.dec.com

Abstract

General-purpose operating systems provide inadequate support for resource management in large-scale servers. Applications lack sufficient control over scheduling and management of machine resources, which makes it difficult to enforce priority policies, and to provide robust and controlled service. There is a fundamental mismatch between the original design assumptions underlying the resource management mechanisms of current general-purpose operating systems, and the behavior of modern server applications. In particular, the operating system's notions of protection domain and *resource principal* coincide in the process abstraction. This coincidence prevents a process that manages large numbers of network connections, for example, from properly allocating system resources among those connections.

We propose and evaluate a new operating system abstraction called a *resource container*, which separates the notion of a protection domain from that of a resource principal. Resource containers enable fine-grained resource management in server systems and allow the development of robust servers, with simple and firm control over priority policies.

1 Introduction

Networked servers have become one of the most important applications of large computer systems. For many users, the perceived speed of computing is governed by server performance. We are especially interested in the performance of Web servers, since these must often scale to thousands or millions of users.

Operating systems researchers and system vendors have devoted much attention to improving the performance of Web servers. Improvements in operating system performance have come from reducing data movement costs [2, 35, 43], developing better kernel algorithms for protocol control block (PCB) lookup [26] and file descriptor allocation [6], improving stability under overload [15, 30], and improving server control mechanisms [5, 21]. Application designers have also attacked performance problems by making more efficient use of existing operating systems. For example, while early Web servers used a process per connection, recent servers [41, 49] use a single-process model, which reduces context-switching costs.

While the work cited above has been fruitful, it has generally treated the operating system's application programming interface (API), and therefore its core abstractions, as a constant. This has frustrated efforts to solve thornier problems of server scaling and effective control over resource consumption. In particular, servers may still be vulnerable to "denial of service" attacks, in which a malicious client manages to consume all of the server's resources. Also, service providers want to exert explicit control over resource consumption policies, in order to provide differentiated quality of service (QoS) to clients [1] or to control resource usage by guest servers in a Rent-A-Server host [45]. Existing APIs do not allow applications to directly control resource consumption throughout the host system.

The root of this problem is the model for resource management in current general-purpose operating systems. In these systems, scheduling and resource management primitives do not extend to the execution of significant parts of kernel code. An application has no control over the consumption of many system resources that the kernel consumes on behalf of the application. The explicit resource management mechanisms that do exist are tied to the assumption that a process is what constitutes an independent activity[1]. Processes are the resource principals: those entities between which the resources of the system are to be shared.

Modern high-performance servers, however, often use a single process to perform many independent activities. For example, a Web server may manage hundreds or even thousands of simultaneous network connections, all within the same process. Much of the resource consumption associated with these connections occurs in kernel

[1] We use the term *independent activity* to denote a unit of computation for which the application wishes to perform separate resource allocation and accounting; for example, the processing associated with a single HTTP request.

mode, making it impossible for the application to control which connections are given priority[2].

In this paper, we address resource management in monolithic kernels. While microkernels and other novel systems offer interesting alternative approaches to this problem, monolithic kernels are still commercially significant, especially for Internet server applications.

We describe a new model for fine-grained resource management in monolithic kernels. This model is based on a new operating system abstraction called a *resource container*. A resource container encompasses all system resources that the server uses to perform a particular independent activity, such as servicing a particular client connection. All user and kernel level processing for an activity is charged to the appropriate resource container, and scheduled at the priority of the container. This model allows fairly arbitrary interrelationships between protection domains, threads and resource containers, and can therefore support a wide range of resource management scenarios.

We evaluate a prototype implementation of this model, as a modification of Digital UNIX, and show that it is effective in solving the problems we described.

2 Typical models for high-performance servers

This section describes typical execution models for high-performance Internet server applications, and provides the background for the discussion in following sections. To be concrete, we focus on HTTP servers and proxy servers, but most of the issues also apply to other servers, such as mail, file, and directory servers. We assume the use of a UNIX-like API; however, most of this discussion is valid for servers based on Windows NT.

An HTTP server receives requests from its clients via TCP connections. (In HTTP/1.1, several requests may be sent serially over one connection.) The server listens on a well-known port for new connection requests. When a new connection request arrives, the system delivers the connection to the server application via the `accept()` system call. The server then waits for the client to send a request for data on this connection, parses the request, and then returns the response on the same connection. Web servers typically obtain the response from the local file system, while proxies obtain responses from other servers; however, both kinds of server may use a cache to speed retrieval. Stevens [42] describes the basic operation of HTTP servers in more detail.

The architecture of HTTP servers has undergone radical changes. Early servers forked a new process to handle each HTTP connection, following the classical UNIX model. The forking overhead quickly became a problem, and subsequent servers (such as the NCSA httpd [32]), used a set of pre-forked processes. In this model, shown in Figure 1, a master process accepts new connections and passes them to the pre-forked worker processes.

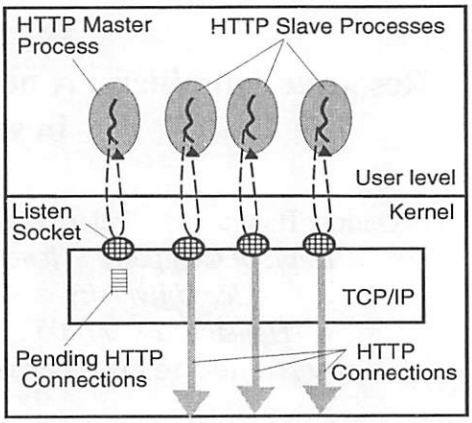


Fig. 1: A process-per connection HTTP server with a master process.

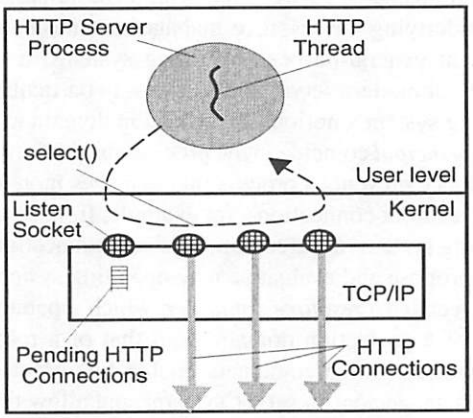


Fig. 2: A single-process event-driven server.

Multi-process servers can suffer from context-switching and interprocess communication (IPC) overheads [11, 38], so many recent servers use a single-process architecture. In the event-driven model (Figure 2), the server uses a single thread to manage all connections at the server. (Event-driven servers designed for multiprocessors use one thread per processor.) The server uses the `select()` (or `poll()`) system call to simultaneously wait for events on all connections it is handling. When `select()` delivers one or more events, the server's main loop invokes handlers for each ready connection. Squid [41] and Zeus [49] are examples of event-driven servers.

Alternatively, in the single-process multi-threaded model (Figure 3), each connection is assigned to a unique thread. These can either be user-level threads or kernel threads. The thread scheduler is responsible for time-sharing the CPU between the various server threads.

[2] In this paper, we use the term *priority* loosely to mean the current scheduling precedence of a resource principal, as defined by the scheduling policy based on the principal's scheduling parameters. The scheduling policy in use may not be priority based.

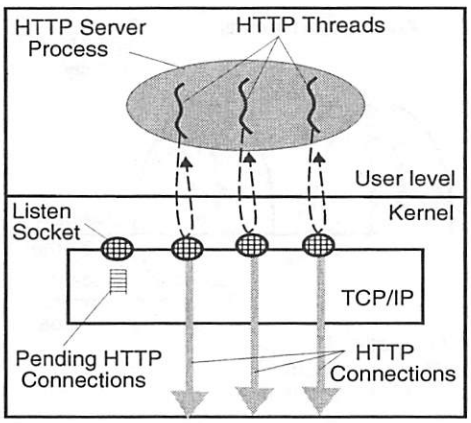


Fig. 3: A single-process multi-threaded server.

Idle threads accept new connections from the listening socket. The *AltaVista* front-end uses this model [8].

So far, we have assumed the use of static documents (or "resources", in HTTP terms). HTTP also supports requests for dynamic resources, for which responses are created on demand, perhaps based on client-provided arguments. For example, a query to a Web search engine such as *AltaVista* resolves to a dynamic resource.

Dynamic responses are typically created by auxiliary third-party programs, which run as separate processes to provide fault isolation and modularity. To simplify the construction of such auxiliary programs, standard interfaces (such as CGI [10] and FastCGI [16]) support communication between Web servers and these programs. The earliest interface, CGI, creates a new process for each request to a dynamic resource; the newer FastCGI allows persistent CGI processes. Microsoft and Netscape have defined library-based interfaces [29, 34] to allow the construction of third-party dynamic resource modules that reside in the main server process, if fault isolation is not required; this minimizes overhead.

In summary, modern high-performance HTTP servers are implemented as a small set of processes. One main server process services requests for static documents; dynamic responses are created either by library code within the main server process, or, if fault isolation is desired, by auxiliary processes communicating via a standard interface. This is ideal, in theory, because the overhead of switching context between protection domains is incurred only if absolutely necessary. However, structuring a server as a small set of processes poses numerous important problems, as we show in the next section.

3 Shortcomings of current resource management models

An operating system's scheduling and memory allocation policies attempt to provide fairness among resource principals, as well as graceful behavior of the system under various load conditions. Most operating systems treat a process, or a thread within a process, as the schedulable entity. The process is also the "chargeable" entity for the allocation of resources, such as CPU time and memory.

A basic design premise of such process-centric systems is that a process is the unit that constitutes an independent activity. This give the process abstraction a dual function: it serves both as a protection domain and as a resource principal. As protection domains, processes provide isolation between applications. As resource principals, processes provide the operating system's resource management subsystem with accountable entities, between which the system's resources are shared.

We argue that this equivalence between protection domains and resource principals, however, is not always appropriate. We will examine several scenarios in which the natural boundaries of resource principals do not coincide with either processes or threads.

3.1 The distinction between scheduling entities and activities

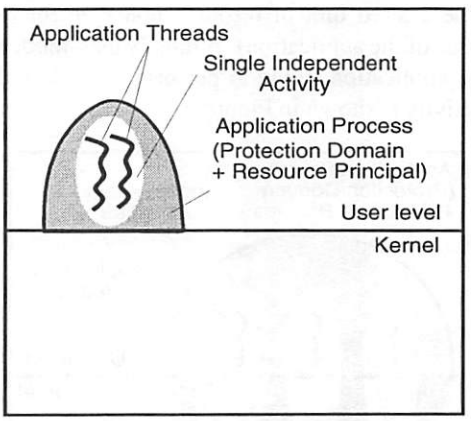


Fig. 4: A classical application.

A classical application uses a single process to perform an independent activity. For such applications, the desired units of isolation and resource consumption are identical, and the process abstraction suffices. Figure 4 shows a mostly user-mode application, using one process to perform a single independent activity.

In a network-intensive application, however, much of the processing is done in the kernel. The process is the correct unit for protection isolation, but it does not encompass all of the associated resource consumption; in most operating systems, the kernel generally does not control or properly account for resources consumed during the processing of network traffic. Most systems do protocol processing in the context of software interrupts, whose execution is either charged to the unlucky process running at the time of the interrupt, or to no process at all. Figure 5 shows the relationship between the application, process, resource principal and independent activity entities for a network-intensive application.

Some applications are split into multiple protection

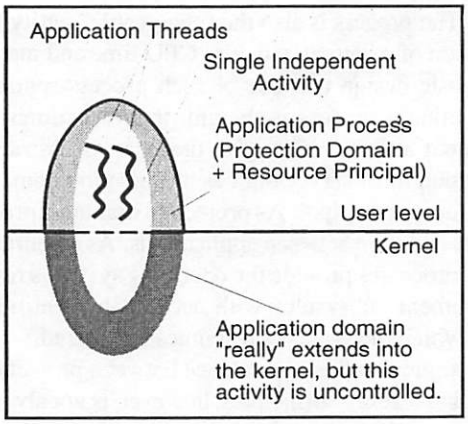


Fig. 5: A classical network-intensive application.

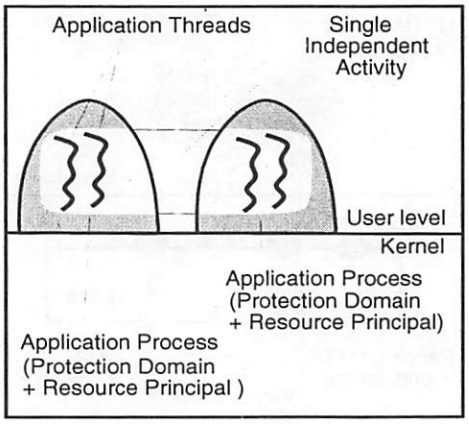


Fig. 6: A multi-process application.

domains (for example, to provide fault isolation between different components of the application). Such applications may still perform a single independent activity, so the desired unit of protection (the process) is different from the desired unit of resource management (all the processes of the application). A mostly user-mode multi-process application trying to perform a single independent activity is shown in Figure 6.

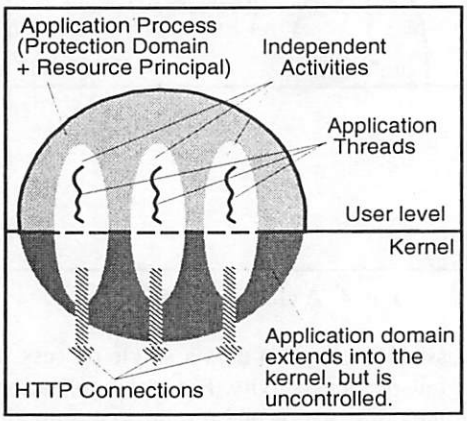


Fig. 7: A single-process multi-threaded server.

In yet another scenario, an application consists of a single process performing multiple independent activities. Such applications use a single protection domain, to reduce context-switching and IPC overheads. For these applications, the correct unit of resource management is smaller than a process: it is the set of all resources being used by the application to accomplish a single independent activity. Figure 7 shows, as an example, a single-process multi-threaded Internet server.

Real-world single-process Internet servers typically combine the last two scenarios: a single process usually manages all of server's connections, but additional processes are employed when modularity or fault isolation is necessary (see section 2). In this case, the desired unit of resource management includes part of the activity of the main server process, and also the entire activity of, for example, a CGI process.

In some operating systems, e.g., Solaris, threads assume some of the role of a resource principal. In these systems, CPU usage is charged to individual threads rather than to their parent processes. This allows threads to be scheduled either independently, or based on the combined CPU usage of the parent process's threads. The process is still the resource principal for the allocation of memory and other kernel resources, such as sockets and protocol buffers.

We stress that it is not sufficient to simply treat threads as the resource principals. For example, the processing for a particular connection (activity) may involve multiple threads, not always in the same protection domain (process). Or, a single thread may be multiplexed between several connections.

3.2 Integrating network processing with resource management

As described above, traditional systems provide little control over the kernel resources consumed by network-intensive applications. This can lead to inaccurate accounting, and therefore inaccurate scheduling. Also, much of the network processing is done as the result of interrupt arrivals, and interrupts have strictly higher priority than any user-level code; this can lead to starvation or livelock [15, 30]. These issues are particularly important for large-scale Internet servers.

Lazy Receiver Processing (LRP) [15] partially solves this problem, by more closely following the process-centric model. In LRP, network processing is integrated into the system's global resource management. Resources spent in processing network traffic are associated with and charged to the application process that caused the traffic. Incoming network traffic is processed at the scheduling priority of the process that received the traffic, and excess traffic is discarded early. LRP systems exhibit increased fairness and stable overload behavior.

LRP extends a process-centered resource principal into the kernel, leading to the situation shown in Fig-

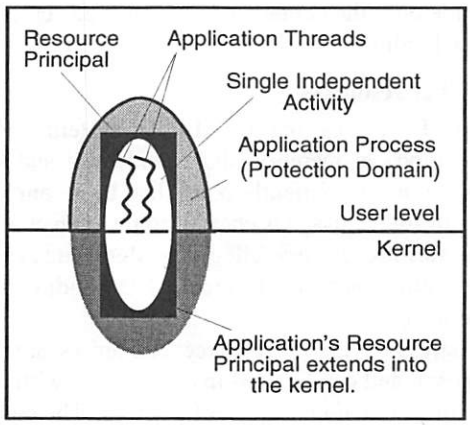


Fig. 8: A network-intensive application in a LRP system.

ure 8. However, LRP maintains the equivalence between resource principal and process; it simply makes it more accurate. LRP, by itself, does not solve all of the problems that arise when the process is not the correct unit of resource management.

3.3 Consequences of misidentified resource principals

Our fundamental concern is to allow an application to explicitly allocate resource consumption among the independent activities that it manages. This is infeasible if the operating system's view of activity differs from that of the application, or if the system fails to account for large chunks of consumption. Yet it is crucial for a server to support accurately differentiated QoS among its clients, or to prevent overload from denial-of-service attacks, or to give its existing connections priority over new ones.

With a single-process server, for example, traditional operating systems see only one resource principal – the process. This prevents the application from controlling consumption of kernel CPU time (and other kernel resources) by various network connections *within* this resource principal. The application cannot control the order in which the kernel delivers its network events; nor, in most systems, can it control whether it receives network events before other processes do.

It is this lack of a carefully defined concept of resource principal, independent from other abstractions such as process or thread, that precludes the application control we desire.

4 A new model for resource management

To address the problems of inadequate control over resource consumption, we propose a new model for fine-grained resource management in monolithic kernels. We introduce a new abstraction, called a *resource container*, for the operating system's resource principal.

Sections 4.1 through 4.7 describe the resource container model in detail. Section 4.8 then discusses its use in Internet servers.

4.1 Resource containers

A resource container is an abstract operating system entity that logically contains all the system resources being used by an application to achieve a particular independent activity. For a given HTTP connection managed by a Web server, for example, these resources include CPU time devoted to the connection, and kernel objects such as sockets, protocol control blocks, and network buffers used by the connection.

Containers have attributes; these are used to provide scheduling parameters, resource limits, and network QoS values. A practical implementation would require an access control model for containers and their attributes; space does not permit a discussion of this issue.

The kernel carefully accounts for the system resources, such as CPU time and memory, consumed by a resource container. The system scheduler can access this usage information and use it to control how it schedules threads associated with the container; we discuss scheduling in detail in Section 4.3. The application process can also access this usage information, and might use it, for example, to adjust the container's numeric priority.

Current operating systems, as discussed in Section 3, implicitly treat processes as the resource principals, while ignoring many of the kernel resources they consume. By introducing an explicit abstraction for resource containers, we make a clear distinction between protection domains and resource principals, and we provide for fuller accounting of kernel resource consumption. This provides the flexibility necessary for servers to handle complex resource management problems.

4.2 Containers, processes, and threads

In classical systems, there is a fixed association between threads and resource principals (which are either the threads themselves, or the processes containing the threads). The resource consumption of a thread is charged to the associated resource principal, and this information is used by the system when scheduling threads.

With resource containers, the binding between a thread and a resource principal is dynamic, and under the explicit control of the application; we call this the thread's *resource binding*. The kernel charges the thread's resource consumption to this container. Multiple threads, perhaps from multiple processes, may simultaneously have their resource bindings set to a given container.

A thread starts with a default resource container binding (inherited from its creator). The application can rebind the thread to another container as the need arises. For example, a thread time-multiplexed between several connections changes its resource binding as it switches from handling one connection to another, to ensure correct accounting of resource consumption.

4.3 Resource containers and CPU scheduling

CPU schedulers make their decisions using information about both the desired allocation of CPU time, and the recent history of actual usage. For example, the traditional UNIX scheduler uses numeric process priorities (which indicate desired behavior) modified by time-decayed measures of recent CPU usage; lottery scheduling [48] uses lottery tickets to represent the allocations. In systems that support threads, the allocation for a thread may be with respect only to the other threads of the same process ("process contention scope"), or it may be with respect to all of the threads in the system ("system contention scope").

Resource containers allow an application to associate scheduling information with an activity, rather than with a thread or process. This allows the system's scheduler to provide resources directly to an activity, no matter how it might be mapped onto threads.

The container mechanism supports a large variety of scheduling models, including numeric priorities, guaranteed CPU shares, or CPU usage limits. The allocation attributes appropriate to the scheduling model are associated with each resource container in the system. In our prototype, we implemented a multi-level scheduling policy that supports both fixed-share scheduling and regular time-shared scheduling.

A thread is normally scheduled according to the scheduling attributes of the container to which it is bound. However, if a thread is multiplexed between several containers, it may cost too much to reschedule it (recompute its numeric priority and decide whether to preempt it) every time its resource binding changes. Also, with a feedback-based scheduler, using only the current container's resource usage to calculate a multiplexed thread's numeric priority may not accurately reflect its recent usage. Instead, the thread should be scheduled based on the *combined* resource allocations and usage of all the containers it is currently handling.

To support this, our model defines a binding, called a *scheduler binding*, between each thread and the set of containers over which it is currently multiplexed. A priority-based scheduler, for example, would construct a thread's scheduling priority from the combined numeric priorities of the resource containers in its scheduler binding, possibly taking into account the recent resource consumption of this set of containers.

A thread's scheduler binding is set implicitly by the operating system, based on the system's observation of the thread's resource bindings. A thread that services only one container will therefore have a scheduler binding that includes just this container. The kernel prunes the scheduler binding set of a container, periodically removing resource containers that the thread has not recently had a resource binding to. In addition, an application can explicitly reset a thread's scheduler binding to include only the container to which it currently has a resource binding.

4.4 Other resources

Like CPU cycles, the use of other system resources such as physical memory, disk bandwidth and socket buffers can be conveniently controlled by resource containers. Resource usage is charged to the correct activity, and the various resource allocation algorithms can balance consumption between principals depending on specific policy goals.

We stress here that resource containers are just a mechanism, and can be used in conjunction with a large variety of resource management policies. The container mechanism causes resource consumption to be charged to the correct principal, but does not change what these charges are. Unfortunately, policies currently deployed in most general-purpose systems are able to control consumption of resources other than CPU cycles only in a very coarse manner, which is typically based on static limits on total consumption. The development of more powerful policies to control the consumption of such resources has been the focus of complimentary research in application-specific paging [27, 20, 24] and file caching [9], disk bandwidth allocation [46, 47], and TCP buffer management [39].

4.5 The resource container hierarchy

Resource containers form a hierarchy. The resource usage of a child container is constrained by the scheduling parameters of its parent container. For example, if a parent container is guaranteed at least 70% of the system's resources, then it and its child containers are collectively guaranteed 70% of the system's resources.

Hierarchical resource containers make it possible to control the resource consumption of an entire subsystem without constraining (or even understanding) how the subsystem allocates and schedules resources among its various independent activities. For example, a system administrator may wish to restrict the total resource usage of a Web server by creating a parent container for all the server's resource containers. The Web server can create an arbitary number of child containers to manage and distribute the resources allocated to its parent container among its various independent activities, e.g. different client requests.

The hierarchical structure of resource containers makes it easy to implement fixed-share scheduling classes, and to enforce a rich set of priority policies. Our prototype implementation supports a hierarchy of resource principals, but only supports resource bindings between threads and leaf containers.

4.6 Operations on resource containers

The resource container mechanism includes these operations on containers:

Creating a new container: A process can create a new resource container at any time (and may have multiple containers available for its use). A default resource container is created for a new process as part of a `fork()`, and the first thread of the new process is bound to this container. Containers are visible to the application as file descriptors (and so are inherited by a new process after a `fork()`).

Set a container's parent: A process can change a container's parent container (or set it to "no parent").

Container release: Processes release their references to containers using `close()`; once there are no such descriptors, and no threads with resource bindings, to the container, it is destroyed. If the parent P of a container C is destroyed, C's parent is set to "no parent."

Sharing containers between processes: Resource containers can be passed between processes, analogous to the transfer of descriptors between UNIX processes (the sending process retains access to the container). When a process receives a reference to a resource container, it can use this container as a resource context for its own threads. This allows an application to move or share a computation between multiple protection domains, regardless of the container inheritance sequence.

Container attributes: An application can set and read the attributes of a container. Attributes include scheduling parameters, memory allocation limits, and network QoS values.

Container usage information: An application can obtain the resource usage information charged to a particular container. This allows a thread that serves multiple containers to timeshare its execution between these containers based on its particular scheduling policy.

These operations control the relationship between containers, threads, sockets, and files:

Binding a thread to a container: A process can set the resource binding of a thread to a container at any time. Subsequent resource usage by the thread is charged to this resource container. A process can also obtain the current resource binding of a thread.

Reset the scheduler binding: An application can reset a thread's scheduler binding to include only its current resource binding.

Binding a socket or file to a container: A process can bind the descriptor for a socket or file to a container; subsequent kernel resource consumption on behalf of this descriptor is charged to the container. A descriptor may be bound to at most one container, but many descriptors may be bound to one container. (Our prototype currently supports binding only sockets, not disk files.)

4.7 Kernel execution model

Resource containers are effective only if kernel processing on behalf of a process is performed in the resource context of the appropriate container. As discussed in Section 3, most current systems do protocol processing in the context of a software interrupt, and may fail to charge the costs to the proper resource principal.

LRP, as discussed in Section 3.2, addresses this problem by associating arriving packets with the receiving process as early as possible, which allows the kernel to charge the cost of received-packet processing to the correct process. We extend the LRP approach, by associating a received packet with the correct resource container, instead of with a process. If the kernel uses threads for network processing, the thread handling a network event can set its resource binding to the resource container; a non-threaded kernel might use a more ad-hoc mechanism to perform this accounting.

When there is pending protocol processing for multiple containers, the priority (or other scheduling parameters) of these containers determines the order in which they are serviced by the kernel's network implementation.

4.8 The use of resource containers

We now describe how a server application can use resource containers to provide robust and controlled behavior. We consider several example server designs.

First, consider a single-process multi-threaded Web server, that uses a dedicated kernel thread to handle each HTTP connection. The server creates a new resource container for each new connection, and assigns one of a pool of free threads to service the connection. The application sets the thread's resource binding to the container. Any subsequent kernel processing for this connection is charged to the connection's resource container. This situation is shown in Figure 9.

If a particular connection (for example, a long file transfer) consumes a lot of system resources, this consumption is charged to the resource container. As a result, the scheduling priority of the associated thread will decay, leading to the preferential scheduling of threads handling other connections.

Next, consider an event-driven server, on a uniprocessor, using a single kernel thread to handle all of its connections. Again, the server creates a new resource container for each new connection. When the server does processing for a given connection, it sets the thread's resource binding to that container. The operating system adds each such container to the thread's scheduler bind-

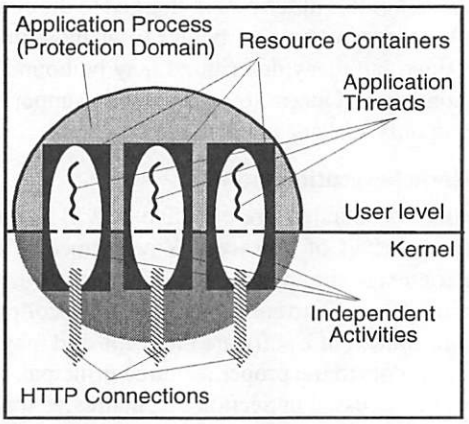


Fig. 9: Containers in a multi-threaded server.

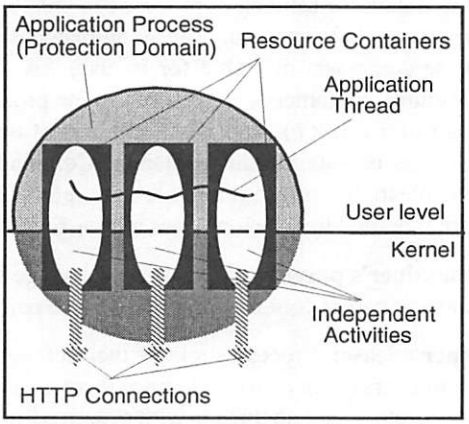


Fig. 10: Containers in an event-driven server.

ing. Figure 10 depicts this situation.

If a connection consumes a lot of resources, this usage is charged to the corresponding container. The server application can obtain this usage information, and use it both to adjust the container's numeric priority, and to control how it subsequently expends its resources for the connection.

Both kinds of servers, when handling a request for a dynamic (CGI) document, pass the connection's container to the CGI process. This may either be done by inheritance, for traditional CGI using a child process, or explicitly, when persistent CGI server processes are used. (If the dynamic processing is done in a module within the server process itself, the application simply binds its thread to the appropriate container.)

A server may wish to assign different priorities to requests from different sources, even for processing that occurs in the kernel before the application sees the connection. This could be used to defend against some denial-of-service attacks, and could also be used by an ISP to provide an enhanced class of service to users who have paid a premium.

To support this prioritization, we define a new `sockaddr` namespace that includes a "filter" specifying a set of foreign addresses, in addition to the usual Internet address and port number. Filters are specified as tuples consisting of a template address and a CIDR network mask [36]. The application uses the `bind()` system call to bind multiple server sockets, each with the same *<local-address, local-port>* tuple but with a different *<template-address, CIDR-mask>* filter. The system uses these filters to assign requests from a particular client, or set of clients, to the socket with a matching filter. By associating a different resource container with each socket, the server application can assign different priorities to different sets of clients, prior to listening for and accepting new connections on these sockets. (One might also want to be able to specify complement filters, to accept connections *except* from certain clients.)

The server can use the resource container associated with a listening socket to set the priority of accepting new connections relative to servicing the existing ones. In particular, to defend against a denial-of-service attack from a specific set of clients, the server can create a socket whose filter matches this set, and then bind it to a resource container with a numeric priority of zero. (This requires the network infrastructure to reject spoofed source addresses, a problem currently being addressed [33].)

A server administrator may wish to restrict the total CPU consumption of certain classes of requests, such as CGI requests, requests from certain hosts, or requests for certain resources. The application can do this by creating a container for each such class, setting its attributes appropriately (e.g., limiting the total CPU usage of the class), and then creating the resource container for each individual request as the child of the corresponding class-specific container.

Because resource containers enable precise accounting for the costs of an activity, they may be useful to administrators simply for sending accurate bills to customers, and for use in capacity planning.

Resource containers are in some ways similar to many resource management mechanisms that have been developed in the context of multimedia and real-time operating systems [17, 19, 22, 28, 31]. Resource containers are distinguished from these other mechanism by their generality, and their direct applicability to existing general purpose operating systems. See Section 6 for more discussion of this related work.

5 Performance

We performed several experiments to evaluate whether resource containers are an effective way for a Web server to control resource consumption, and to provide robust and controlled service.

5.1 Prototype implementation

Our prototype was implemented as modifications to the Digital UNIX 4.0D kernel. We changed the CPU

scheduler, the resource management subsystem, and the network subsystem to understand resource containers.

We modified Digital UNIX's CPU scheduler scheduler to treat resource containers as its resource principals. A resource container can obtain a fixed-share guarantee from the scheduler (within the CPU usage restrictions of its parent container), or can choose to time-share the CPU resources granted to its parent container with its sibling containers. Fixed-share guarantees are ensured for timescales that are in the order of tens of seconds or larger. Containers with fixed-share guarantees can have child containers; time-share containers cannot have children. In our prototype, threads can only be bound to leaf-level containers.

We changed the TCP/IP subsystem to implement LRP-style processing, treating resource containers as resource principals. A per-process kernel thread is used to perform processing of network packets in priority order of their containers. To ensure correct accounting, this thread sets its resource binding appropriately while processing each packet.

Implementing the container abstraction added 820 lines of new code to the Digital UNIX kernel. About 1730 lines of kernel code were changed and 4820 lines of code were added to integrate containers as the system's resource principals, and to implement LRP-style network processing. Of these 6550 lines (1730 + 4820) of integration code, 2342 lines (142 changed, 2200 new) concerned the CPU scheduler, 2136 lines (205 changed, 1931 new) were in the network subsystem, and the remainder were spread across the rest of the kernel.

Code changes were small for all the server applications that we considered, though they were sometimes fairly pervasive throughout the application.

5.2 Experimental environment

In all experiments, the server was a Digital Personal Workstation 500au (500Mhz 21164, 8KB I-cache, 8KB D-cache, 96KB level 2 unified cache, 2MB level 3 unified cache, SPECint95 = 12.3, 128MB of RAM), running our modified version of Digital UNIX 4.0D. The client machines were 166MHz Pentium Pro PCs, with 64MB of memory, and running FreeBSD 2.2.5. All experiments ran over a private 100Mbps switched Fast Ethernet.

Our server software was a single-process event-driven program derived from thttpd [44]. We started from a modified version of thttpd with numerous performance improvements, and changed it to optionally use resource containers. Our clients used the S-Client software [4].

5.3 Baseline throughput

We measured the throughput of our HTTP server running on the unmodified kernel. When handling requests for small files (1 KByte) that were in the filesystem cache, our server achieved a rate of 2954 requests/sec. using connection-per-request HTTP, and 9487 requests/sec. using persistent-connection HTTP. These rates saturated the CPU, corresponding to per-request CPU costs of 338μs and 105μs, respectively.

5.4 Costs of new primitives

We measured the costs of primitive operations on resource containers. For each new primitive, a user-level program invoked the system call 10,000 times, measured the total elapsed time, and divided to obtain a mean "warm-cache" cost. The results, in Table 1, show that all such operations have costs much smaller than that of a single HTTP transaction. This implies that the use of resource containers should add negligible overhead.

Operation	Cost (μs)
create resource container	2.36
destroy resource container	2.10
change thread's resource binding	1.04
obtain container resource usage	2.04
set/get container attributes	2.10
move container between processes	3.15
obtain handle for existing container	1.90

Table 1: Cost of resource container primitives.

We verified this by measuring the throughput of our server running on the modified kernel. In this test, the Web server process created a new resource container for each HTTP request. The throughput of the system remained effectively unchanged.

5.5 Prioritized handling of clients

Our next experiment tested the effectiveness of resource containers in enabling prioritized handling of clients by a Web server. We consider a scenario where a server's administrator wants to differentiate between two classes of clients (for example, based on payment tariffs).

Our experiment used an increasing number of low-priority clients to saturate a server, while a single high-priority client made requests of the server. All requests were for the same (static) 1KB file, with one request per connection. We measured the response time perceived by the high-priority client.

Figure 11 shows the results. The y-axis shows the response time seen by the high-priority client (T_{high}) as a function of the number of concurrent low-priority clients. The dotted curve shows how (T_{high}) varies when using the unmodified kernel. The application attempted to give preference to requests from the high-priority client by handling events on its socket, returned by `select()`, before events on other sockets. The figures shows that, despite this preferential treatment, (T_{high}) increases sharply when there are enough low-priority clients to saturate the server. This happens because most of request processing occurs inside the kernel, and so is uncontrolled.

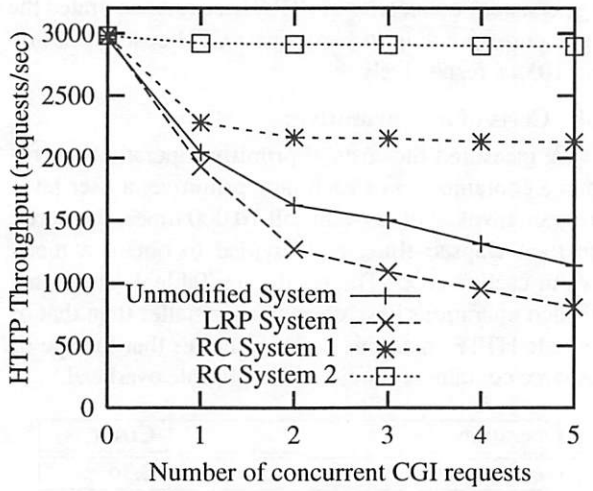


Fig. 12: Throughput with competing CGI requests.

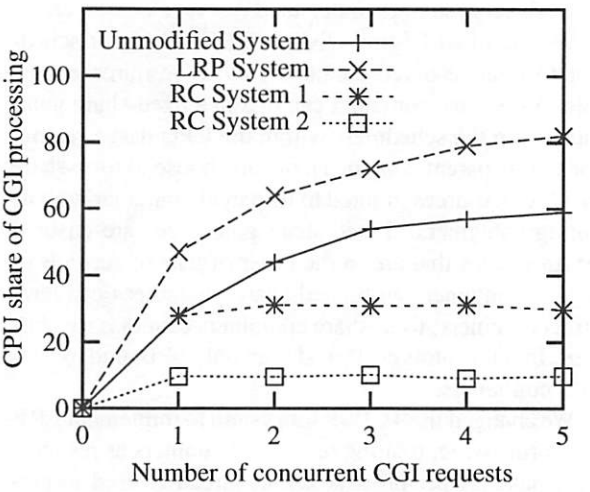


Fig. 13: CPU share of CGI requests.

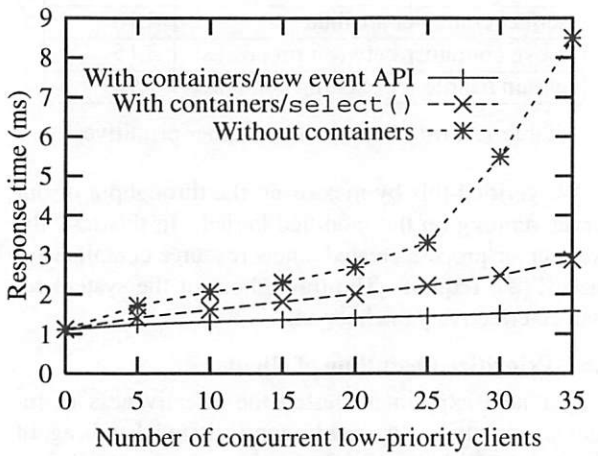


Fig. 11: How T_{high} varies with load.

The dashed and the solid curve in Figure 11 shows the effect of using resource containers. Here, the server uses two containers, with different numeric priorities, assigning the high-priority requests to one container, and the low-priority requests to another. The dashed curve, labeled "With containers/`select()`", shows the effect of resource containers with the application still using `select()` to wait for events. T_{high} increases much less than in the original system. Resource containers allow the application to control resource consumption at almost all levels of the system. For example, TCP/IP processing, which is performed in FIFO order in classical systems, is now performed in priority order.

The remaining increase in response time is due to some known scalability problems of the `select()` system call [5, 6]. These problems can be alleviated by a smart implementation described in [6], but some inefficiency is inherent to the semantics of the `select()` API. The problem is that each call to `select()` must specify, via a bitmap, the complete set of descriptors that the application is interested in. The kernel must check the status of each descriptor in this set. This causes overhead linear in the number of descriptors handled by the application.

The solid curve, labeled "With containers/new event API", shows the variation in T_{high} when the server uses a new scalable event API, described in [5]. In this case, T_{high} increases very slightly as the number of low-priority clients increases. The remaining slight increase in T_{high} reflects the cost of packet-arrival interrupts from low-priority connections. The kernel must handle these interrupts and invoke a packet filter to determine the priority of the packet.

5.6 Controlling resource usage of CGI processing

Section 2 described how requests for dynamic resources are typically handled by processes other than the main Web server process. In a system that time-shares the CPU equally between processes, these back-end (CGI) processes may gain an excessive share of the CPU, which reduces the throughput for static documents. We constructed an experiment to show how a server can use resource containers to explicitly control the CPU costs of CGI processes.

We measured the throughput of our Web server (for cached, 1 KB static documents) while increasing the number of concurrent requests for a dynamic (CGI) resource. Each CGI request process consumed about 2 seconds of CPU time. These results are shown in the curve labeled "Unmodified System" in Figure 12.

As the number of concurrent CGI requests increases, the CPU is shared among a larger set of processes, and the main Web server's share decreases; this sharply reduces the throughput for static documents. For example, with only 4 concurrent CGI requests, the Web server

itself gets only 40% of the CPU, and the static-request throughput drops to 44% of its maximum.

The main server process actually gets slightly more of the CPU than does each CGI process, because of misaccounting for network processing. This is shown in Figure 13, which plots the total CPU time used by all CGI processes.

In Figures 12 and 13, the curves labeled "LRP System" show the performance of an LRP version of Digital UNIX. LRP fixes the misaccounting, so the main server process shares the CPU equally with other processes. This further reduces the throughput for static documents.

To measure how well resource containers allow fine-grained control over CGI processes, we modified our server so that each container created for a CGI request was the child of a specific "CGI-parent" container. This CGI-parent container was restricted to a maximum fraction of the CPU (recall that this restriction includes its children). In Figures 12 and 13, the curves labeled "RC System 1" show the performance when the CGI-parent container was limited to 30% of the CPU; the curves labeled "RC System 2" correspond to a limit of 10%.

Figure 13 shows that the CPU limits are enforced almost exactly. Figure 12 shows that this effectively forms a "resource sand-box" around the CGI processes, and so the throughput of static requests remains almost constant as the number of concurrent CGI requests increases from 1 to 5.

Note that the Web server could additionally impose relative priorities among the CGI requests, by adjusting the resource limits on each corresponding container.

5.7 Immunity against SYN-flooding

We constructed an experiment to determine if resource containers, combined with the filtering mechanism described in Section 4.7, allow a server to protect against denial-of-service attacks using "SYN-flooding." In this experiment, a set of "malicious" clients sent bogus SYN packets to the server's HTTP port, at a high rate. We then measured the server's throughput for requests from well-behaved clients (for a cached, 1 KB static document).

Figure 14 shows that the throughput of the unmodified system falls drastically as the SYN-flood rate increases, and is effectively zero at about 10,000 SYNs/sec. We modified the kernel to notify the application when it drops a SYN (due to queue overflow). We also modified our server to isolate the misbehaving client(s) to a low-priority listen-socket, using the filter mechanism described in Section 4.8. With these modifications, even at 70,000 SYNs/sec., the useful throughput remains at about 73% of maximum. This slight degradation results from the interrupt overhead of the SYN flood. Note that LRP, in contrast to our system, cannot protect against such SYN floods; it cannot filter traffic to a given port based on the source address.

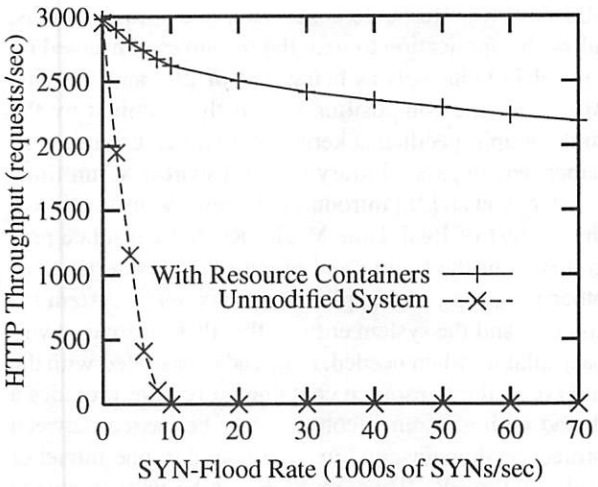


Fig. 14: Server behavior under SYN-flooding attack.

5.8 Isolation of virtual servers

Section 5.6 shows how resource containers allow "resource sand-boxes" to be put around CGI processes. This approach can be used in other applications, such as controlling the total resource usage of guest servers in a Rent-A-Server [45] environment.

In current operating systems, each guest server, which might consist of many processes, can appear to the system as numerous resource principals. The number may vary dynamically, and has little relation to how much CPU time the server's administrator wishes to allow each guest server.

We performed an informal experiment to show how resource containers solve this problem. We created 3 top-level containers and restricted their CPU consumption to fixed CPU shares. Each container was then used as the root container for a guest server. Subsequently, three sets of clients placed varying request loads on these servers; the requests included CGI resources. We observed that the total CPU time consumed by each guest server exactly matched its allocation. Moreover, because the resource container hierarchy is recursive, each guest server can itself control how its allocated resources are re-divided among competing connections.

6 Related Work

Many mechanisms have been developed to support fine-grained research management. Here, we contrast these with our resource container abstraction.

The Scout operating system [31] is based on the *path* abstraction, representing an I/O channel (such as a TCP connection) through a multi-layered system. A path encapsulates the specific attributes of an I/O channel, and allows access to these attributes across layers. Paths have been used to implement fine-grained resource management in network appliances, including Web server ap-

pliances [40]. Resource containers, in contrast to paths, allow the application to treat the resources consumed by several I/O channels as being part of the same activity. Moreover, the composition of a path is limited by the router graph specified at kernel-build time; resource containers encompass arbitrary sets of resources at run-time.

Mercer et al. [28] introduced the *reserve* abstraction in the context of Real-Time Mach. Reserves insulate programs from the timing and execution characteristics of other programs. An application can reserve system resources, and the system ensures that these resources will be available, when needed, to threads associated with the reserve. Like a resource container, a reserve provides a thread with a resource context, may be passed between protection domains, and may be bound to one thread or multiple threads. Thus, reserves can be used to charge to one resource principal the resources consumed by an activity distributed across protection domains. Unlike resource containers, reserves neither account for, nor control, kernel-mode processing on behalf of an activity (RT Mach is a microkernel system, so network processing is done in user mode [25]). Moreover, resources containers can be structured hierarchically and can manage system resources other than CPU.

The *activity* abstraction in Rialto [22] is similar to resource containers. Like a resource container, an activity can account for resource consumption both across protection domains and at a granularity smaller than a protection domain. However, Rialto is an experimental real-time object-oriented operating system and was designed from scratch for resource accountability. In contrast to Scout, RT Mach and Rialto, our work aimed at developing a resource accounting mechanism for traditional UNIX systems with minimal disruption to existing APIs and implementations.

The *migrating threads* of Mach [17] and AlphaOS [13], and the *shuttles* of Spring [19] allow the resource consumption of a thread (or a shuttle) performing a particular independent activity to be charged to the correct resource management entity, even when the thread (or shuttle) moves across protection domains. However, these systems do not separate the concepts of thread and resource principal, and so cannot correctly handle applications in which a single thread is associated with multiple independent activities, such as an event-driven Web server. Mach and Spring are also microkernel systems, and so do not raise the issue of accounting for kernel-mode network processing.

The *reservation domains* [7] of Eclipse and the *Software Performance Units* of Verghese et al. [46] allow the resource consumption of a group of processes to be considered together for the purpose of scheduling. These abstractions allow a resource principal to encompass a number of protection domains; unlike resource containers, neither abstraction addresses scenarios, such a single-process Web server, where the natural extent of a resource principal is more complicated.

A number of mainframe operating systems [14, 37, 12] provide resource management at a granularity other than a process. These systems allow a group of processes (e.g. all processes owned by a given user) to be treated as a single resource principal; in this regard, they are similar to resource containers. Unlike our work, however, there are no provisions for resource accounting at a granularity smaller than a process. These systems account and limit the resources consumed by a process group over long periods of time (on the order of hundreds of minutes or longer). Resource containers, on the other hand, can support policies for fine-grained, short-term resource scheduling, including real-time policies.

The resource container hierarchy is similar to other hierarchical structures described in the scheduling literature [18, 48]. These hierarchical scheduling algorithms are complementary to resource containers, and could be used to schedule threads according to the resource container hierarchy.

The exokernel approach [23] gives application software as much control as possible over raw system resources. Functions implemented by traditional operating systems are instead provided in user-mode libraries. In a network server built using an exokernel, the application controls essentially all of the protocol stack, including the device drivers; the storage system is similarly exposed. The application can therefore directly control the resource consumption for all of its network and file I/O. It seems feasible to implement the resource container abstraction as a feature of an exokernel library operating system, since the exokernel delegates most resource management to user code.

Almeida et al. [1] attempted to implement QoS support in a modified Apache [3] Web server, running on a general-purpose monolithic operating system. Apache uses a process for each connection, and so they mapped QoS requirements onto numeric process priorities, experimenting both with a fully user-level implementation, and with a slightly modified Linux kernel scheduler. They were able to provide differentiated HTTP service to different QoS classes. However, the effectiveness of this technique was limited by their inability to control kernel-mode resource consumption, or to differentiate between existing connections and new connection requests. Also, this approach does not extend to event-driven servers.

Several researchers have studied the problem of controlling kernel-mode network processing. Mogul and Ramakrishnan [30] improved the overload behavior of a busy system by converting interrupt-driven processing into explicitly-scheduled processing. Lazy Receiver Processing (LRP) [15] extended this by associating received packets as early as possible with the receiving process, and then performed their subsequent processing based

on that process's scheduling priority. Resource containers generalize this idea, by separating the concept of a resource principal from that of a protection domain.

7 Conclusion

We introduced the resource container, an operating system abstraction to explicitly identify a resource principal. Resource containers allow explicit and fine-grained control over resource consumption at all levels in the system. Performance evaluations demonstrate that resource containers allow a Web server to closely control the relative priority of connections and the combined CPU usage of various classes of requests. Together with a new `sockaddr` namespace, resource containers provide immunity against certain types of denial of service attacks. Our experience suggests that containers can be used to address a large variety of resource management scenarios beyond servers; for instance, we expect that container hierarchies are effective in controlling resource usage in multi-user systems and workstation farms.

Acknowledgments

We are grateful to Deborah Wallach, Carl Waldspurger, Willy Zwaenepoel, our OSDI shepherd Mike Jones, and the anonymous reviewers, whose comments have helped to improve this paper. This work was supported in part by NSF Grants CCR-9803673, CCR-9503098, and by Texas TATP Grant 003604.

References

[1] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing Differentiated Quality-of-Service in Web Hosting Services. In *Proc. Workshop on Internet Server Performance*, June 1998.

[2] E. W. Anderson and J. Pasquale. The Performance of the Container Shipping I/O System. In *Proc. Fifteenth ACM Symposium on Operating System Principles*, Dec. 1995.

[3] Apache. http://www.apache.org/.

[4] G. Banga and P. Druschel. Measuring the Capacity of a Web Server. In *Proc. 1997 USENIX Symp. on Internet Technologies and Systems*, Dec. 1997.

[5] G. Banga, P. Druschel, and J. C. Mogul. Better operating system features for faster network servers. In *Proc. Workshop on Internet Server Performance*, June 1998. Condensed version appears in ACM SIGMETRICS *Performance Evaluation Review* 26(3):23–30, Dec. 1998.

[6] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proc. 1998 USENIX Technical Conference*, June 1998.

[7] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proc. 1998 USENIX Technical Conference*, June 1998.

[8] M. Burrows. Personal communication, Mar. 1998.

[9] P. Cao. *Application Controlled File Caching and Prefetching*. PhD thesis, Princeton University, Jan. 1996.

[10] The Common Gateway Interface. http://hoohoo.ncsa.uiuc.edu/cgi/.

[11] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proc. 1996 USENIX Technical Conference*, Jan. 1996.

[12] D. Chess and G. Waldbaum. The VM/370 resource limiter. *IBM Systems Journal*, 20(4):424–437, 1981.

[13] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An Architectural Overview of The Alpha Real-Time Distributed Kernel. In *Workshop on Micro-Kernels and Other Kernel Architectures*, Apr. 1992.

[14] P. Denning. Third generation computer systems. *ACM Computing Surveys*, 3(4):175–216, Dec. 1971.

[15] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Oct. 1996.

[16] Open Market. FastCGI Specification. http://www.fastcgi.com/.

[17] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proc. 1994 Winter USENIX Conference*, Jan. 1994.

[18] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Oct. 1996.

[19] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proc. 1993 Summer USENIX Conference*, June 1993.

[20] K. Harty and D. R. Cheriton. Application-Controlled Physical Memory using External Page-Cache Replacement. In *Proc. of the* 5th *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992.

[21] J. C. Hu, I. Pyrali, and D. C. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Proc. 2nd Global Internet Conf.*, Nov. 1997.

[22] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Bar-

rera. Modular real-time resource management in the Rialto operating system. In *Proc. 5th Workshop on Hot Topics in Operating Systems*, May 1995.

[23] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Janotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proc. 16th Symp. on Operating System Principles*, Oct. 1997.

[24] K. Krueger, D. Loftesness, A. Vahdat, and T. Anderson. Tools for the Development of Application-Specific Virtual Memory Management. In *Proc. 8th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 1993.

[25] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol processing in real-time Mach. In *In Proc. IEEE Real-time Technology and Applications Symp.*, June 1996.

[26] P. E. McKenney and K. F. Dove. Efficient Demultiplexing of Incoming TCP Packets. In *Proceedings of the SIGCOMM '92 Conference*, Aug. 1993.

[27] D. McNamee and K. Armstrong. Extending the Mach External Pager Interface to Accomodate User-Level Page Replacement Policies. In *Proc. USENIX Mach Symp.*, Oct. 1990.

[28] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proc. of the IEEE Int'l Conf. on Multimedia Computing and Systems*, May 1994.

[29] Microsoft Corporation ISAPI Overview. http://www.microsoft.com/msdn/sdk/platforms/doc/sdk/internet/src/isapimrg.htm.

[30] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. *ACM Trans. on Computer Systems*, 15(3):217–252, Aug. 1997.

[31] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Oct. 1996.

[32] NCSA httpd. http://hoohoo.ncsa.uiuc.edu/.

[33] North American Network Operators Group (NANOG). Mailing List Archives, Thread #01974. http://www.merit.edu/mail.archives/html/nanog/threads.html#01974, Apr. 1998.

[34] Netscape Server API. http://www.netscape.com/newsref/std/server_api.html.

[35] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proc. 3rd Symp. on Operating Systems Design and Implementation*, Feb. 1999.

[36] Y. Rekhter and T. Li. An Architecture for IP Address Allocation with CIDR. RFC 1518, Sept. 1993.

[37] R. Schardt. An MVS tuning approach (OS problem solving). *IBM Systems Journal*, 19(1):102–119, 1980.

[38] S. E. Schechte and J. Sutaria. A Study of the Effects of Context Switching and Caching on HTTP Server Performance. http://www.eecs.harvard.edu/~stuart/Tarantula/FirstPaper.html.

[39] J. Semke and J. M. M. Mathis. Automatic TCP Buffer Tuning. In *Proc. SIGCOMM '98 Conference*, Sept. 1998.

[40] O. Spatscheck and L. L. Petersen. Defending Against Denial of Service Attacks in Scout. In *Proc. 3rd Symp. on Operating Systems Design and Implementation*, Feb. 1999.

[41] Squid. http://squid.nlanr.net/Squid/.

[42] W. Stevens. *TCP/IP Illustrated Volume 3*. Addison-Wesley, Reading, MA, 1996.

[43] M. N. Thadani and Y. A. Khalidi. An efficient zero-copy I/O framework for UNIX. Technical Report SMLI TR-95-39, Sun Microsystems Laboratories, Inc., May 1995.

[44] thttpd. http://www.acme.com/software/thttpd/.

[45] A. Vahdat, E. Belani, P. Eastham, C. Yoshikawa, T. Anderson, D. Culler, and M. Dahlin. WebOS: Operating System Services For Wide Area Applications. In *Proc. Seventh Symp. on High Performance Distributed Computing*, July 1998.

[46] B. Verghese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proc.* 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Oct. 1998.

[47] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Managament*. PhD thesis, Massachusetts Institute of Technology, Sept. 1995.

[48] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proc. Symp. on Operating Systems Design and Implementation*, Nov. 1994.

[49] Zeus. http://www.zeus.co.uk/.

Defending Against Denial of Service Attacks in Scout

Oliver Spatscheck
Department of Computer Science
University of Arizona

Larry L. Peterson
Department of Computer Science
Princeton University

Abstract

We describe a two-dimensional architecture for defending against denial of service attacks. In one dimension, the architecture accounts for all resources consumed by each I/O path in the system; this accounting mechanism is implemented as an extension to the path object in the Scout operating system. In the second dimension, the various modules that define each path can be configured in separate protection domains; we implement hardware enforced protection domains, although other implementations are possible. The resulting system—which we call Escort—is the first example of a system that simultaneously does end-to-end resource accounting (thereby protecting against resource based denial of service attacks where principals can be identified) and supports multiple protection domains (thereby allowing untrusted modules to be isolated from each other). The paper describes the Escort architecture and its implementation in Scout, and reports a collection of experiments that measure the costs and benefits of using Escort to protect a web server from denial of service attacks.

1 Introduction

It is becoming increasingly important that networked computing systems be able to protect themselves from denial of service attacks. For example, a web server needs to be able to detect and defend itself from an attacker that is consuming its resources by trying to initiate TCP connection establishment as rapidly as possible—the so called SYN attack [22]. Protecting against denial of service attacks involves three steps:

Accounting: A necessary first step is to account for all resources consumed by every principal.

Detection: A denial of service attack is detected when the resources consumed by a given principal exceed those allowed by some system policy.

Containment: Once an attack is detected, it must be possible to reclaim the consumed resources using as few additional resources as possible, otherwise, removal of an offending principal becomes a denial of service attack in its own right.

Attacks on traditional operating systems like Unix [18] frequently exploit the lack of accounting within the kernel, that is, before the work has been assigned to a particular user (principal). For example, it is possible for an attacker to consume all available TCP ports before a single message is dispatched to a user process which implements the policy and could detect the attack. Even if an attack is detected, it is often difficult, if not impossible, to reclaim all the resources consumed by the offending principal. Consider, for example, something as commonplace as a distributed file system: cached file blocks, NFS mount points, device buffers, and network connection state almost always have a longer lifetime than the user process that requested them. There is no direct way to account such resources towards a principal, and certainly no way to reclaim them when the principal is removed from the system because it has violated some usage policy.

Recent multimedia operating systems like Scout and Nemesis [13, 14] begin to address this problem by isolating data streams and minimizing cross talk between streams; cross talk is resource contention that interferes with the system's ability to make quality-of-service guarantees to each stream. Although these systems are successful in isolating streams, they do not provide the fine-grain accounting of resource usage needed to detect denial of service attacks. They are also limited in that their isolation mechanisms do not span multiple protection domains; they assume all resources used by a given data stream are confined to a single domain. Assuming a single protection domain is unrealistically restrictive, for example, it precludes a web server from running untrusted CGI scripts.

This situation points to a dilemma faced in designing a secure system: how to simultaneously support protection

domains that allow untrusted components of the system to be isolated from each other, yet account for all system resources consumed (potentially across multiple domains) by a single principal. This paper addresses this dilemma by making two contributions. First, it presents a fine-grain resource accounting mechanism that has been implemented in the Scout operating system. The mechanism is able to account for virtually 100% of the resources used by a given principal at a low overhead of 8%. Second, it describes how this mechanism can be made to work across multiple protection domains. The paper does not offer any novel denial of service policies, but it does describe a working web server based on this architecture, and measures its performance while enforcing a representative set of usage policies.

The limitation of this work is that it is impossible to charge a piece of work to a particular principal until the principal has been identified. For incoming network packets, this means the system is vulnerable from the time a packet arrives until it has been demultiplexed and authenticated. The architecture we describe takes two steps to minimize the impact of this window of vulnerability. First, it pushes the demultiplexing/authentication decision as early as possible. Exactly how early depends on the protocols being used and the environment in which the system exists. For example, a WWW server using IPSEC [1] can authenticate an IPv6 datagrams cheaply using a secure hash function. This happens during demultiplexing, earlier than it would be possible using TLS [7]. In another example, a WWW server positioned behind a filtering router might use IP addresses from the local network for authentication, trusting the router to filter inappropriate datagrams. In a third, and more complex environment, IP addresses could be rated by an intrusion detection system, with resources allocated according to the trustworthiness of those addresses. In all three cases, it is important that the OS does not architecturally force a late demultiplexing decision.

The second way our architecture minimizes the impact of late authentication is that, even when the system has not yet determined precisely what principal is responsible for a particular packet, certain classes of packets can be aggregated and given only limited resources. For example, IPSEC allows early authentication by requiring a key exchange protocol to establish a shared key. In such an environment, the server is vulnerable to an attack by a new client that consumes server resources by sending the server bogus key exchange requests. Our architecture allows the WWW server to give preference to clients that already posses valid shared keys, thereby maintaining connectivity to the current set of clients while under attack. We will demonstrate this feature in a later section, showing how a web server might limit the cycles spent processing new connections (e.g., SYN packets) by giving preference to existing connections.

2 Architecture

This section defines Scout's security architecture. It begins with an overview of Scout, and then describes how we have extended Scout to support both fine-grain accounting and protection domains. It concludes with a brief discussion of how the resulting system—which we call Escort—facilitates the enforcement of different security policies.

2.1 Configurability

Modules are the unit of program development and configurability in Scout. Each Scout module provides a well-defined and independent function. Well-defined means that there is usually either a standard interface specification, or some existing practice that defines the exact function of a module. Independent means that each single module provides a useful, self-contained service. That is, the module should not depend on there being other specific modules connected to it. Typical examples are modules that implement networking protocols, such as HTTP, IP, UDP, or TCP; modules that implement storage system components, such as VFS, UFS, or SCSI; and modules that implement drivers for the various device types in the system.

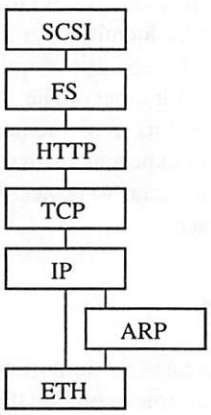


Figure 1: Example Scout Module Graph

To form a complete system, individual modules are connected into a *module graph*: the nodes of the graph correspond to the modules included in the system, and the edges denote the dependencies between these modules. Two modules can be connected by an edge if they support a common *service interface*. These interfaces are typed and enforced by Scout. By configuring Scout with different collections of modules, we can configure kernels for different purposes, including network-attached

devices, web and file servers, firewalls and routers, and multimedia displays. For example, Figure 1 shows an extract of the module graph for a Scout kernel that implements a web server. The configuration includes device drivers for the network and disk devices (ETH and SCSI), four conventional network protocols (ARP, IP, TCP and HTTP), and a simple file system (FS). Such a configuration is specified at build time, and a set of configuration tools assemble the corresponding modules into an executable kernel.

2.2 Path Abstraction

Scout adds a communication-oriented abstraction—the *path*—to the configurable system just described. Intuitively, a path can be viewed as a logical channel through a modular system over which I/O data flows. In other words, the path abstraction defines a channel over which data moves through the system, for example, from input device to output device. Each path is an object that encapsulates two important elements: (1) it defines the sequence of code modules that are applied to the data as it moves through the system, and (2) it represents the entity that is scheduled for execution.

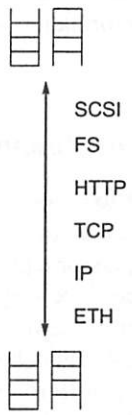


Figure 2: Example HTTP Path

Although the module graph is defined at system build time, paths are created and destroyed at run time as I/O connections are opened and closed. Figure 2 schematically depicts a path that traverses the module graph shown in Figure 1; it has source queues and sink queues, and is labeled with the sequence of software modules that define how the path "transforms" the data it carries. This particular path processes incoming HTTP requests by fetching web pages from disk.

The path-specific local state of each module is stored in a data structure called a *stage*. Stages from a sequence of modules are combined to form the path. In addition to this path-specific state, when executing code within a certain module, paths also have access to the state of the module. For example, a path executing code of the IP module has access to the routing tables stored in the IP module.

Each path goes through three phases during its lifetime. The first phase is path creation, during which the topology of the path—i.e., the sequence of modules it traverses—is determined, and the state of the path is initialized. Path creation is triggered by a pathCreate call to the kernel; the kernel limits path creation according to an access control list (ACL) specified by the system designer.

Specifically, the pathCreate operation takes six arguments: a set of attributes, the starting module, a subject, a subject class, the calling protection domain, and the calling owner. The first four arguments are explicitly given, while the last two are implicitly known from the calling thread. The attribute set defines invariants for the path, such as the port number and IP address for the peer. The kernel uses these invariants, plus the starting module, to determine the path's topology—the sequence of modules that the path traverses. Because only a certain small number of path topologies are useful in a given configuration, it is accurate to think of this process as determining the path's *type* (e.g., an "HTTP path"). Next, the kernel consults the ACL to determine if the entity trying to create the path is allowed to create a path of this type, and if so, what resource limits might be imposed on it. The entity creating the path is identified by the last four arguments to pathCreate: the subject (think of this as a user or a role), a subject class (this defines the availability level [16]), the calling protection domain (see Section 2.3), and the calling owner (see Section 2.4). At this point, the path exists and its resource limits are known.

Then the path enters its second phase, during which data is sent and received over it. Both send and receive work in the obvious way: data is enqueued at one end of the path and a thread is scheduled to execute the path. There is one complication, however. When data arrives on a device—e.g., a network packet arrives on the Ethernet—the kernel must determine to which path it belongs. This is done in a way that is analogous to path creation: the kernel identifies the path incrementally by invoking a demux operation on a sequence of modules. Each module's demux function has three choices: (1) it can determine that a unique path has not yet been identified and call the demux function of some adjacent module; (2) it can reject the request and drop the data; or (3) it can return a unique path. The demux function is side-effect free.

The last phase of a path is invoked by a pathDestroy or pathKill call to the kernel. In case of pathDestroy the kernel invokes a destroy function associated with each module along the path in the same order in which they were initialized before it frees all resources used by the path. pathKill frees all the path's resources, but does not invoke the destroy functions.

2.3 Protection Domains

Escort extends the basic Scout architecture by isolating the modules that have been configured into the system into separate protection domains. The kernel—which implements the path operations described above, as well as other objects described in the next section—runs in a privileged protection domain. The protection domain that each module is to run in is specified at configuration time. Trusted modules can be placed in the privileged domain. Modules can also be multiply instantiated, both across different protection domains, and in the same protection domain.

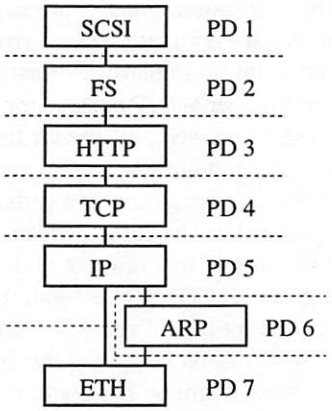


Figure 3: Modules Partitioned into Protection Domains

Figure 3 shows the module graph for our example web server partitioned into separate protection domains; one module per domain in this example. (The device drivers also have access to the memory regions used to access their devices.) This configuration represents the maximum possible separation. A less restrictive configuration might, for example, combine TCP, IP and ARP within one protection domain.

In addition to the kernel and the set of modules configured into the system, Escort also supports libraries that implement commonly used functions. Library code is trusted by their users, and so is mapped executable into all protection domains. Escort currently supplies libraries to manage messages, hash tables, participant addresses, attributes, queues, heaps, and time. It also includes a standard C library.

The current version of Escort runs in a single 64-bit address space and implements protection domains using hardware mechanisms available on the Alpha microprocessor. Modules not linked into the privileged domain invoke kernel services using a hardware trap. However, software fault isolation [24], type safe languages like Java, and proof carrying code [15] could be used instead.

Since the code for each module and library might be used by multiple protection domains, the calling environment for a given invocation of a library or module function must be specified. Furthermore, since modules can also be multiply instantiated within one protection domain, it is not sufficient to have one data segment per protection domain. Therefore, Escort explicitly passes the calling environment as the first argument to any procedure, optimizing for stateless libraries and libraries that access only protection domain state. This is similar to the approach described in [19].

Each module supports a well-known initialization function. When an Escort system boots, the kernel initializes every module by switching to the appropriate protection domain and calling the init function on each module in that domain. The modules initialize their global state and create an initial set of paths.

Finally, we return to the issue of demultiplexing incoming network packets, but this time in light of multiple protection domains. The base demux mechanism in Scout trusts the demux functions contributed by each module. Although not yet implemented in Escort, alternative mechanisms—e.g., pattern-based demultiplexers like PathFinder [3] and the current system augmented with Proof Carrying Code [15]—would be more appropriate since they do not trust the demultiplexing code to be correct and to not leak information via the demultiplexing decision.

2.4 Accounting for Resource Usage

A key goal of Escort is to account for all resource usage. Towards this end, all resources are charged to an *owner*, which can be either a path or a protection domain. Paths are the preferred choice since they most naturally correspond to the actual user of the resources. However, there are certain resources that cannot be accounted to a particular path. For example, an IP routing table cannot be directly associated with (charged to) any individual IP flow; the memory used by the routing table is associated with the protection domain that runs the IP module.

There are only a few differences between protection domains and paths in terms of ownership. One is that protection domains have a heap and paths do not. The reason for this is that the kernel allows memory allocation at the page level only. For paths this is extremely inefficient since it would require a path to allocate at least one page for each protection domain it crosses. To keep the accounting mechanism accurate, the protection domain can charge paths that cross it with memory usage. The memory charged toward a path is then deducted from the memory charged to the protection domain. In other words, the kernel gives memory pages to protection domains, which in turn implement a heap and hand out smaller memory objects to paths that traverse them.

To allow the automatic reclamation of this memory—and other resources like the reservation of a TCP port—all modules can register destructor functions with a path. This function is called in the module's protection domain when a path is destroyed or killed, and results in charge for the memory being transfered back to the protection domain. The destructor function usually frees all memory charged toward the path. However, the domain is ultimately responsible for the freeing of the memory, that is, returning the page back to the kernel.

Another difference is that paths can be destroyed without destroying the modules or protection domains they cross. However, if a protection domain is destroyed, all paths crossing that protection domain are also destroyed. This is necessary since paths can access the global state of all modules they cross and this state will be removed if the protection domain is destroyed. For example after destroying the protection domain containing the IP module, IP's routing table will no longer be accessible by paths anymore.

```
struct Owner {
    OwnerType type; /* PATH or PD */
    /* Accounting */
    u_long kmem;
    u_long pages;
    u_long IoBuffer,
    u_long threads;
    u_long stacks;
    u_long cycle;
    u_long events;
    u_long semaphores;
    /* Tracking */
    PageList  pages;
    ThreadList threads;
    IoBufferLockList iobufferlock;
    EventList  event;
    SemaphoreList semaphore;
    /* Scheduling */
    Scheduler scheduler;
    /* Resource Monitoring */
    Resource limits;
};
```

Figure 4: Owner Data Structure

Figure 4 shows the Owner data structure; this structure is the first element of both the path and protection domain data structures. The Owner structure is divided into three parts. The first part keeps a count of the resources—kernel memory, memory pages, IOBuffer, threads, stacks, CPU cycles, events, and semaphores—used by this owner. The fields in this part are used to decide if the resource part of the security policy has been violated. Note that the kmem field counts the amount of memory used to store the kernel objects referenced in the second part of the data structure.

The second part contains doubly linked lists of the actual kernel objects associated with this owner; these objects are described in Section 3. These lists support the fast removal of the corresponding objects in event that the owner must be destroyed. The Scheduler object contains the information necessary to schedule threads belonging to this owner. The exact contents of this data structure depends on the scheduler used. The last part contains the resource limits of the owner. This object is more fully described in Section 2.5.

Whenever a new resource is requested, the owner is explicitly passed as an argument to the kernel allocator. Although not mandated by the architecture, many policies require that this argument must match the owner of the current thread.

2.5 Specifying Resource Limits

Owners are charged for resources they use, with any limits placed on this usage specified at system configuration time (for protection domains), and at path creation time (for paths). The resource limits for a particular owner are given by the Resource object of the Owner data structure. The action to be taken when a given limit is exceeded is specified in the Limit object; possible actions include destroying the path, denying the request, or preventing further demultiplexing of incoming data to the path. Figure 5 shows both the Resource and Limit data structures.

```
struct Limit {
    int val;
    Action action
};
struct Resource {
    Id subject;
    Id subject_class;
    Limit kmem;
    Limit pages;
    Limit IoBuffer;
    Limit threads;
    Limit stacks;
    Limit cycle;
    Limit events;
    Limit semaphores;
    Limit yield;
    Limit attribute[attr_count];
};
```

Figure 5: Limit and Resource Data Structures.

The Resource object defines limits for the very same resources as accounted for in the Owner object: kernel

memory, pages, IOBuffers, threads, stacks, cycle, events, semaphores and attributes. In addition, the yield field limits the maximum number of cycles a thread can run without yielding the processor, attr_count is a system constant limiting the number of attributes which can be associated with a path and the attribute field limits the values of those attributes. The Resource object also contains identifiers for subjects, which correspond to users or roles and subject classes which represent availability levels in multi level availability systems. These identifiers are used to aggregate resource usage over multiple paths.

All resource limits, except for the yield and cycle restrictions, are enforced by a resource monitor. This monitor is called whenever resources are allocated or freed, or when attributes change. The resource monitor is also responsible for monitoring aggregated resource utilization for subjects and subject classes according to a given policy. To support multiple policies, Escort allows the appliance designer to configure different resource monitors into the system. Currently, Escort uses a simple resource monitor that compares the resources used against the stated limit, and performs the appropriate action when the limit is exceeded. It does not support aggregation of resources.

The yield and cycle restrictions are enforced directly by the kernel at clock interrupt time, and if a violation of policy occurs, the only action allowed is to destroy the associated owner.

2.6 Remarks

Although we have been focusing on how Escort accounts for resource usage, it is useful to place Escort's security mechanisms in a larger context. Specifically, Escort allows the system designer to enforce a security policy on four different levels.

- The kernel uses a conventional role-based ACL [2] to guard against unauthorized access. The role is determined by the owner of the thread and the current protection domain.
- The module graph defines the base channels of communication between protection domains, and therefore limits information flow between protection domains and those channels.
- The path object allows the system to always charge actions towards the principal that is ultimately responsible for them. Paths also allow us to perform certain complex access control decisions at path creation time instead of path execution time. In this way, a path is similar to a cache of capabilities for a specific owner, and as a consequence, the path creation process becomes an important part of the policy.
- It is possible to configure *filters* between modules in the module graph. Syntactically, filters are just like any other module, except their purpose is to enforce policy rather than to implement a specific function. For example, a filter between TCP and IP might restrict the TCP/IP interface from one that supports "receive packets" to one that supports only "receive packets to port 80". The filter enforces this more restricted interface by filtering data that does not adhere to this restriction. Such filters can be used along with a vanilla TCP module, and conversely, the same TCP module can be flanked by different filters. The important point is that the security policy need not be embedded in the TCP module.

3 Implementation

Escort currently implements 52 system calls that provide access to the following kernel objects: paths, IObuffers, threads, events, semaphores, memory pages, devices, and the console. This section describes the implementation of the first three of these objects in more detail.

3.1 Paths

As already described, paths are created and destroyed using pathCreate, pathDestroy and pathKill operations. The kernel also provides functions that allow data to be enqueued on either end of a path.

```
struct Path {
    struct Owner owner;
    Hash   allowed_pd_crossings;
    StageList stages;
    Queues[4] q;
    ThreadPool t;
    u_long refCnt;
};
```

Figure 6: Path Data Structure

The path data structure, as shown in Figure 6, is accessible only from within the kernel. It contains the owner state, a hash table of allowed protection domain crossings for this path, a list of the stages belonging to the path, pointers to the path input and output queues, a thread pool that provides threads for the path, and a reference counter used to delay pathDestroy but not pathKill calls.

The stages contained in the stage list represent the contribution of each module to the path. Stages communicate using predefined interfaces. The entry point of these interfaces are established during path creation and stored in the map of allowed protection domain crossings. Escort

currently supports interfaces for asynchronous I/O, name resolution, and file access.

3.2 Threads

Threads, like any other resource in Escort, are owned by either a protection domain or a path. This means that the lifetime of a thread is bound by the lifetime of its owner, and as a consequence, threads cannot directly migrate between owners. Keep in mind that the motivation for migrating threads [5] is to allow a single execution context to cross multiple protection domains, but this is already supported in Escort by the explicit path abstraction. In a well designed configuration, thread migration between owners—e.g., from one path to another or from one protection domain to another—should be an uncommon event. Should such a need arise, Escort provides a handoff function that generates a new thread belonging to the target owner. Escort also synchronizes the threads, and wakes up any threads waiting for a thread belonging to an owner that has been destroyed.

Threads owned by a protection domain always execute within this domain and are implemented similar to regular UNIX threads. In contrast, threads owned by a path have the ability to cross the protection domains along the path. These threads have multiple stacks: one for each protection domain in which they can execute, plus a kernel-resident stack that records the protection domains currently being crossed. This is more efficient than assigning a new stack after each protection domain crossing since Escort threads are likely to switch into the same protection domain more than once. For example, a thread used to deliver an ICMP echo request datagram is also used to send the ICMP response, thereby crossing the protection domain containing IP twice.

To call from one domain to another, the call to the target function is executed, resulting in a memory access violation. The kernel then checks to see if the thread is owned by a path, and if the path data structure contains a mapping from the current protection domain to the target environment and function. If this mapping exists, the kernel switches to the appropriate protection domain and continues execution using the same thread. Since the mappings are maintained in a per-path hash table, access time is almost always constant. Upon return, a memory trap to a special address occurs, triggering the kernel to remove the last protection domain crossing from its stack and return to the caller that triggered the protection domain crossing.

Using the Alpha calling conventions, Escort passes integer arguments across protection domain boundaries in registers. Arguments passed by reference are either copied onto the stack that is mapped in the appropriated protection domain, or an IOBuffer (described in section 3.3) is used. This makes inter-domain calls indistinguishable from regular function calls, and allows the system builder to draw protection boundaries between modules as needed. In other words, whether a protection domain boundary sits between any pair of modules need not be known at the time the modules are implemented.

Escort threads cannot be preempted gracefully. They are similar to non-preemptive threads, with the exception that they can be preempted if they are destroyed immediately afterwards. The removal of a thread, however, most likely leaves its owner in an inconsistent state. Therefore, the owner of a removed thread is itself removed. Since Escort allows the kernel to specify a maximum thread runtime without yields for each owner, this mechanism is good enough to deal with runaway threads, but it does not impose the synchronization overhead within modules that would be necessary if preemptive threads were used.

In addition to threadHandoff, threadYield and threadStop operations, the kernel also supports events and semaphores. Again, these objects are owned by either paths or protection domains. Events allow modules to fork new threads that start executing a given function after a specified delay. Semaphores can be used to block threads. The threads that can be blocked on a semaphore are not limited to threads of the owner of the semaphore. If a semaphore is destroyed, however, all threads that do not belong to the owner of the semaphore are unblocked.

The thread scheduler is configured during configuration time. Escort currently supports a priority-based scheduler, a proportional share scheduler, and an EDF scheduler.

3.3 IOBuffers

Escort uses IOBuffers to pass blocks of data between protection domains. IOBuffers are similar to FBufs [8], except they use a more elaborate reference counting scheme and more restrictive mapping rules. IOBuffers are managed by the kernel and can be allocated, locked, unlocked, and associated with an owner. IOBuffers are always allocated as a multiple of the system's page size.

When an IOBuffer is allocated, it is associated with the owner that is specified as an argument. The owner argument is restricted to either the current protection domain, or a path that crosses the current protection domain. If the owner is the current protection domain, the IOBuffer is mapped read/write in that domain. If the IOBuffer is associated with a path, it is mapped read/write in the current protection domain, and read-only in all other protection domains along the path. The current direction that IOBuffer is flowing is also specified as an argument; direction is given by specifying the next stage along the path that will process the IOBuffer.

To allow paths to traverse multiple security levels, it is

possible to designate certain protection domains along a path as termination domains. This limits the read mapping to the protection domains along the path from the current protection domain, up to and including the termination domain. An identifier for the protection domain that can write in an IOBuffer is stored as first long word in the IOBuffer.

The kernel keeps a reference count for each IOBuffer; a buffer's reference count is incremented by locking it. Locking an IOBuffer removes all write privileges from the buffer; this is indicated by setting the protection domain id field in the IOBuffer to zero. The purpose of removing all write permission is that after locking an IOBuffer, the buffer can be checked for consistency and cannot be altered anymore by the original writer.

Unlocking an IOBuffer decrements the reference counter and removes all write mappings. If the reference counter reaches 0, the buffer is freed or added to a buffer cache. If an IOBuffer is allocated, and it has read mappings in the same protection domains as a cached buffer, the current protection domain mapping is changed to read/write and the buffer is reused. The advantage of this scheme is that cached IOBuffers do not have to be cleaned and a buffer allocation requires only changes in one protection domain's memory mapping.

A final kernel call associates a pre-existing IOBuffer with a second owner. The mapping directions and restrictions are specified in the same way as during IOBuffer allocation. This feature is useful for an application that implements a cache (e.g., a web cache): it allows the protection domain that manages the cache to allocate the IOBuffer, and later map the buffer into all protection domains traversed by paths that use (send/receive) the cached data. No copying is required and only one copy of each data item is stored. This association call includes locking the buffer for the second owner. The second owner is also fully charged for the buffer. This is necessary to avoid the case in which the original owner removes its lock and the second owner does not have enough resources to actually own the buffer. The disadvantage is that there are more resources charged for than actually used.

The message library [12] is used to efficiently manage the IOBuffer and offer a simple user interface tailored for manipulating network messages. All meta data used by the message library is stored in IOBuffers. The message library can deal with the possibility that it might lose write permission to an IOBuffer transparently. It also adds another layer of reference counting without involving the kernel. As a result, each protection domain holds at most one kernel lock on any IOBuffer reducing the number of kernel calls.

4 Performance

This section reports measurements of Escort designed to demonstrate the costs and benefits of accounting for resource usage across multiple protection domains. The example system we use for all our experiments is the web server introduced in Section 2.

4.1 Configurations

We measured Escort under a variety of configurations and loads, as outlined below.

4.1.1 Web Server

We tested four configurations of the web server. The first three run on Scout and implement the module graph shown in Figure 1. The fourth configuration runs on Linux. We denote the four configurations as follows:

Scout: All modules and the kernel are configured in a single, privileged protection domain. This configuration does no resource accounting, and so is equivalent to a base Scout kernel.

Accounting: Like Scout, all modules are implemented in a single protection domain, but the system accounts for all resources consumed by paths and protection domains.

Accounting_PD: Includes resource accounting, but each module is configured in its own protection domain. This is the worst-case scenario since each inter-module call implies a protection domain crossing. The module graph for this configuration is shown in Figure 3.

Linux: Apache 1.2.6 web server running on RedHat 5.1 with the 2.0.34 Linux kernel.

4.1.2 Load

The experiments place the following kinds of load on the web server:

Client: A regular client performs a sequence of requests to retrieve the same document. The document sizes used are 1-Byte, 1K-Byte and 10K-Byte. The small document sizes were chosen to minimize the effect of TCP congestion control on the experiment.

QoS Stream: A QoS Stream corresponding to one TCP connection with a guaranteed bandwidth of 1-MBps. A proportional share scheduler is used to ensure that the path responsible for this connection receives this bandwidth. The web server can only guarantee that enough resources for this stream are available on the

server; it cannot guarantee sufficient bandwidth is available within the network.

CGI Attacker: A CGI Attacker performs a GET request at a rate of one every second. The request results in an an infinite-loop thread that emulates a runaway CGI script. This experiment simulates the impact a single user which is allowed to upload CGI scripts on a WWW server can have on the overall performance of the server. It also represents the most basic attack on an active network in which router and end hosts execute code associated with an active packet.

SYN Attacker: A SYN Attacker sends a SYN request to the server at a rate of 1000 every second.

4.1.3 Hardware

All four server configurations, as well as the QoS receiver and the SYN Attacker, run on 300MHz AlphaPC 21064 systems with Digital Fast EtherWORKS PCI 10/100 (DE500) Ethernet adapter connected to a 100Mbps Ethernet. The clients and CGI Attackers run on one to 64 200MHz PentiumPro workstations running Linux. These stations are connected by 100Mbps Ethernet cards to a CISCO Cat5500 switch. The switch is connected by a hub to the web server, the receiver of the QoS stream and the SYN Attacker.

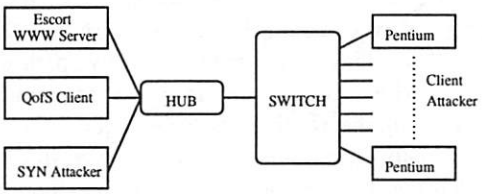


Figure 7: Experimental Setup

The full configuration is shown in Figure 7. There are two reasons for this particular hardware configuration. First, it is possible to run a single Client and a single CGI Attacker on each PentiumPro, eliminating the effects of having overly loaded sources. Second, all Client and CGI Attacker traffic share one 100Mbps Ethernet link. This reduces the number of collisions on the hub and gives the QoS traffic enough network capacity to sustain the 1MBps rate.

4.2 Accounting and Protection Overhead

The first set of experiments measure the overhead imposed on the system by Escort's accounting and protection domain mechanisms. Specifically, Figure 8 reports

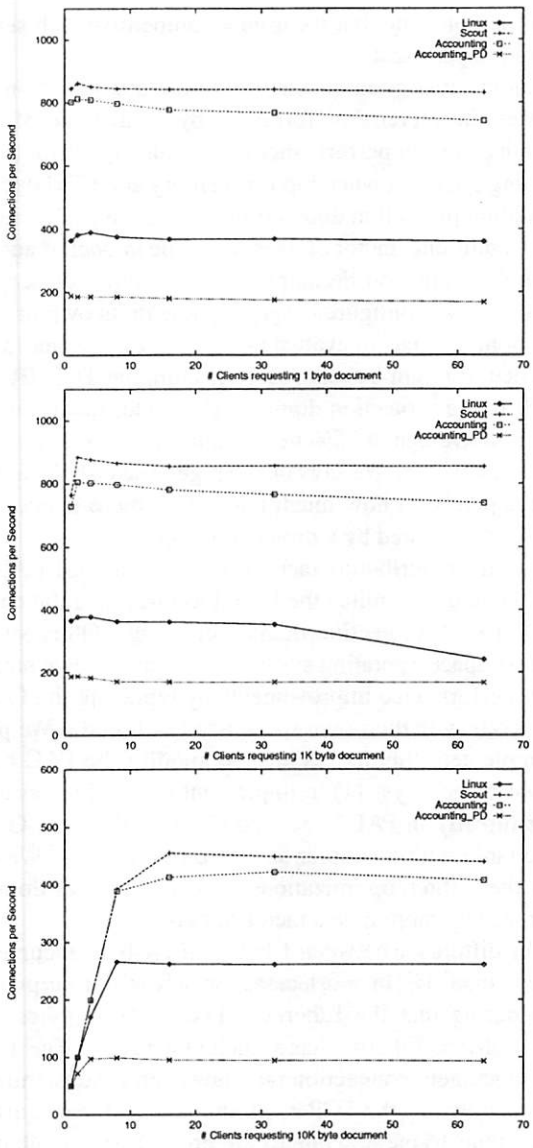


Figure 8: Basic performance of the different configurations in connection per second for a 1Byte document 1KByte document and 10KByte document.

the performance of the web server as it retrieves documents of size 1-byte, 1K-bytes, and 10K-bytes, respectively, from between 1 and 64 parallel clients. All measurements represent the ten-second average measured after the load had been applied for one minute.

The best performance is achieved by the base Scout kernel with Escort's accounting and protection domains disabled; the server is able to handle over two times as many requests as the Apache server running on Linux (800 versus 400 connections per second). This is not surprising considering that Linux is a general-purpose operating system with different design goals. It does, how-

ever, demonstrate that we used a competitive web server for our experiments.

Adding fine-grain accounting to the configuration decreases the server's performance by an average of 8%. This decrease in performance can be mostly attributed to keeping track of ownership for memory and CPU cycles.

Adding protection domains decreases the performance by an additional factor of over four. The impact of adding multiple protection domains is rather high, but keep in mind that we configured every module in its own protection domain so as to evaluate the worst-case scenario. In practice, it might be reasonable to combine TCP, IP, and ETH in one protection domain. Each additional domain adds, on average, a 25% performance penalty to the single domain case. We say "on average" because the actual cost depends on how much interaction there is between modules separated by a protection boundary.

Another contributing factor is a bug in our OSF1 Alpha PAL code that requires the kernel to invalidate the entire TLB at each protection domain crossing. Other single address space operating systems [14] have shown significant performance improvements by replacing the OSF1 PAL code with their own specialized PAL code. We plan to implement this fix, as well as modify the PAL code in two other ways: (1) to implement some of the system calls directly in PAL code, and (2) to replace the OSF1 page table with a simpler structure of our own. We expect these three optimizations to reduce the per-domain overhead by more than a factor of two.

The difference between 1-byte and 1K-byte documents is less than 3% in most cases, which is not surprising considering that the Ethernet MTU is 1460 bytes and our 100Mbps Ethernet has sufficient capacity. The 10K-byte document connection rate, however, is substantially slowed down by the TCP congestion control mechanisms if less than 16 parallel clients are present. If enough parallel clients are present, the connection rate is between 50-60% of the 1K-byte document case. This seems to be a reasonable slowdown to account for sending multiple TCP segments.

4.3 Micro-Experiments

The next set of experiments measure detailed aspects of the architecture.

4.3.1 Accounting Accuracy

Table 1 shows the results of a micro-experiment designed to demonstrate that Escort accounts for all resources consumed during a single HTTP request; here we focus on CPU cycles. The first row (Total Measured) reports the measured number of CPU cycles used during a request for a one-byte document. The measurement starts when the passive path accepts the SYN packet—resulting in the creation of an active path that serves the request—and concludes when the final FIN packet is acknowledged.[1] The next six rows report the total number of cycles accounted for by Escort; the last row (Total Accounted) corresponds to the sum of the preceding five.

We measured two configurations: the second column (**Accounting**) gives the results for a configuration that includes accounting but no protection domains, while the last column (**Accounting_PD**) includes both accounting and protection domains.

Owner	**Accounting**	**Acounting_PD**
Total Measured	402033	1123195
Idle	201493(50%)	9825(1%)
Passive SYN Path	11223(3%)	78882(7%)
Main Active Path	188685(47%)	1033772(92%)
TCP Master Event	38(0%)	514(0%)
Softclock	92 (0%)	200 (0%)
Total Accounted	402031(100%)	1123193(100%)

Table 1: Average number of cycles spent serving 100 serial requests of a one-byte web document.

There are two things to observe about this data. First, Escort accounts for virtually every cycle used, both with and without protection domains. Second, in both the **Accounting** and **Accounting_PD** cases, more then 92% of the non-idle cycles are charged to the active path serving the request. Most of the remaining cycles are accounted to the passive path that receives the SYN request and creates the active path. The number of cycles spent in this passive path is constant for each connection, and therefore its share of the overall time will decrease as the active path does more work.

All other cycles are charged to the TCP master event and the softclock. The TCP master event is responsible for scheduling timeouts of individual TCP connections. The softclock increments the system timer every millisecond and schedules the events. The time spent incrementing the timer and scheduling the softclock is charged to the kernel (it is constant per clock interrupt); the TCP master event is charged to the protection domain that contains TCP; and the cycles spent actually processing each TCP timeout is charged to the path that represents the connection.

[1] Passive and active paths are not an explicit part of the architecture; they are just a way to characterize paths according to their use. The former receive only connection setup messages (e.g., TCP SYN packets), while the latter correspond to open connections on which data messages are sent and received.

4.3.2 Killing a Path

A second micro-experiment measures the time needed to remove all resources associated with a non-cooperating path. In the experiment, a client requests a document and the server enters an endless loop after the GET request is received. Escort then times out the thread after 2ms and destroys the owner.

	Accounting	Accounting_PD	Linux
Cycle	17951	111568	11003

Table 2: Cycle needed to destroy non cooperative path.

Table 2 shows the cycles needed to kill the path from the time the runaway thread is detected until all resources associated with the path in all protection domains are destroyed.

The Linux numbers are measured from the time a parent issues a kill signal until waitpid returns. The Linux number are only reported to give a general idea of the cost of destroying a process and should not be directly compared to the Escort numbers. In Escort, the pathKill operation reclaims all resources, including device buffers and other kernel objects. When protection domains are present, all resources associated with the path in every protection domain—as well as all IPC channels and IOBuffers along the path—are also destroyed. As a point of reference, the 111,568 cycles it takes to reclaim resources in a system with both accounting and protection domains represents approximately 10% of the cycles used to satisfy a single request to retrieve a 1-byte document. These numbers should improve as we optimize the inter-domain calls.

4.4 Defending Against Attacks

We conclude this section by considering three scenarios in which Escort can be used to enforce some resource usage policy. The examples we use were selected to illustrate the impact of policies Escort is able to support. We make no claims that the example policies are strong enough to protect against arbitrary attacks; they are merely representative of policies a system administrator might want to implement.

4.4.1 SYN Attack

The first example is a policy that protects against SYN attacks. We assume that there is a trusted part of the Internet and an untrusted part. The goal is to minimize the impact on HTTP requests from the trusted subnet during a SYN attack from the untrusted subnet.

Escort implements this policy by providing different passive paths: one accepts SYN requests for the trusted subnet and the other from the untrusted subnet. Each passive path uses a path attribute to keep track of the number of active paths it has created which are in the SYN_RECVD state. This path attribute is monitored by the resource monitor and demultiplexing to the passive path is suspended as soon as 64 paths are in the SYN_RECVD state. Therefore, additional SYN requests are identified as such as early as possible and dropped instantly.

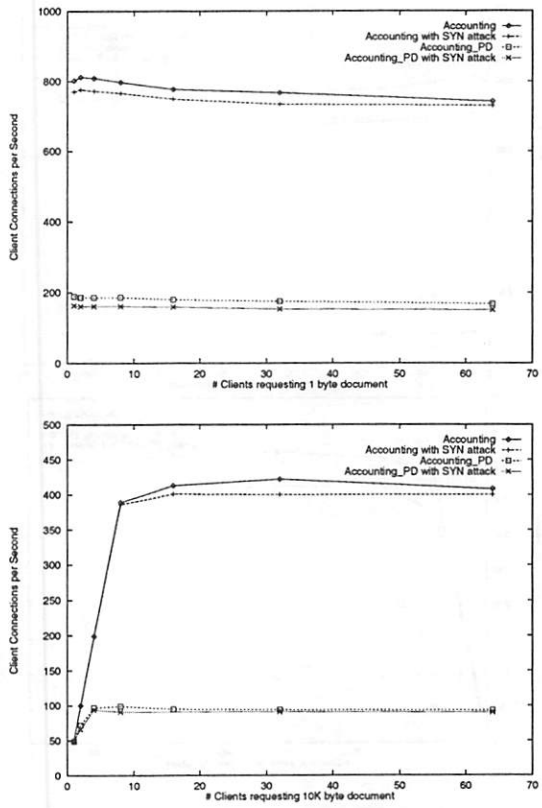


Figure 9: Performance for 1-Byte and 10K-Byte documents for Escort with and without protection domains, with one SYN Attacker generating 1000 SYN requests per second.

Figure 9 shows the impact on the best effort Client traffic of a SYN attack from the untrusted subnet. The best effort traffic of the **Accounting** kernel slows down by less than 5% for both document sizes. The **Accounting_PD** kernel slows down by less than 15%. Both slowdowns are caused by the interrupt handling and demultiplexing time spent on each incoming datagram. The higher slowdown for the **Accounting_PD** kernel is caused by a higher TLB miss rate during demultiplexing. This is because for each domain-crossing, the TLB is invalidated and, therefore, no mappings for demultiplexing are present.

The performance for the 1K-byte documents are not shown but they are within 3% of the 1-byte document.

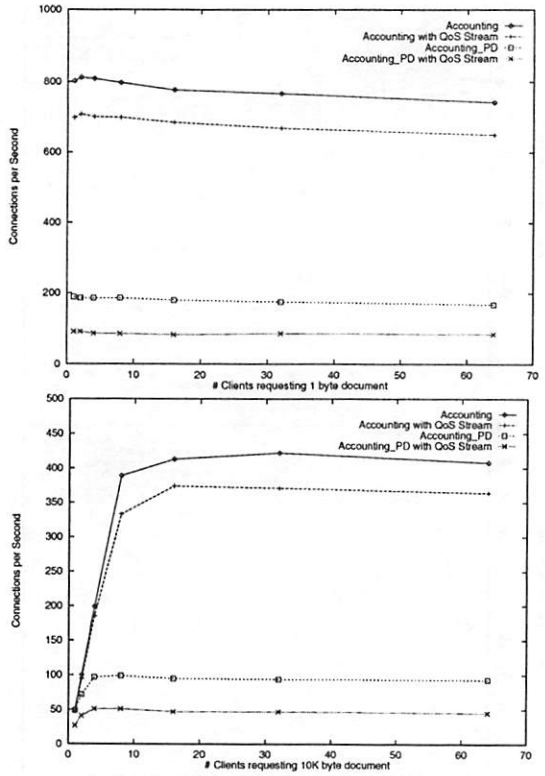


Figure 10: Performance of different configurations with and without a 1MByte/sec Qofs stream in connection per second.

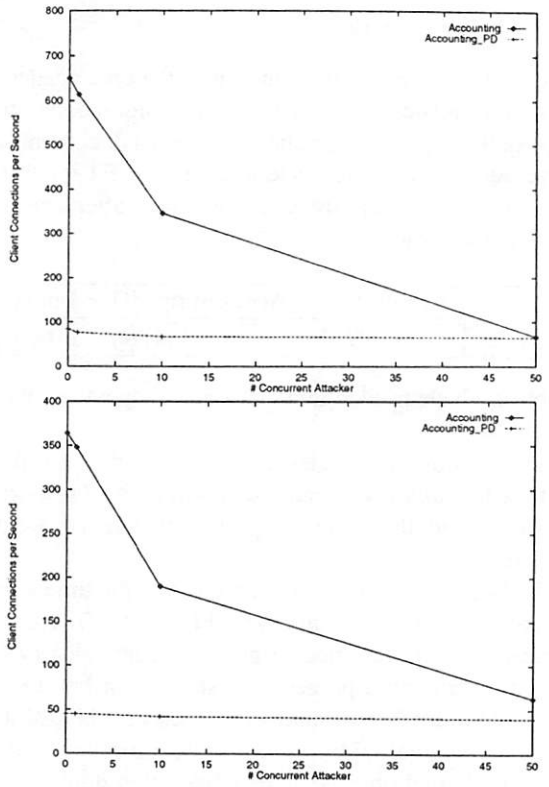


Figure 11: Performance for 1-Byte and 10K-Byte (top down) documents for Escort with and without protection domains, with one 1MBps QoS stream, 64 clients, and a variable number of attackers.

4.4.2 QoS Stream

In the next experiment we add one 1MBps TCP stream to the base experiment described in Section 4.2. The point of this experiment is to demonstrate that Escort is able to sustain a particular quality-of-service request in the face of substantial load. Figure 10 shows the impact on the best effort client traffic with and without protection domains. The results for the 1K-byte document are not shown but are again within 3% of the 1-byte document.

Although not shown in the figure, the ten-second average of the QoS stream is always within 1% of the target rate. The **Accounting** kernel slows down an average of 15%; the **Accounting_PD** kernel slows down by an average of 50%. This is not a surprising result since Escort with protection domains needs substantially more CPU cycles to sustain a 1MBps data stream.

Note that accounting is required to make QoS guarantees, therefore, we are not able to compare Escort with Linux in this case.

4.4.3 CGI Attack

In our final experiment we add 1, 10, or 50 CGI attackers to the previous experiment. As described earlier in this section, each attacker launches one attack per second. Our example policy realizes the attack within 2ms and removes the offending path. As before, we performed this experiment with 1 to 64 clients, document sizes of 1, 1K, and 10K bytes, and a 1MBps guaranteed data stream.

In all cases, the QoS traffic, as measured over ten-second intervals, stays within 1% of the target rate. Since for our example policy we do not distinguish between attackers and clients until the former has used 2ms of CPU time, the system allows connections from attackers with the same probability as from regular clients. This allows the attacker to slow the best effort traffic down substantially since each attacker consumes 2ms worth of CPU cycles before it is detected. This is shown in Figure 11 for the case of 64 concurrent clients. The advantage of Escort in this scenario is that after the attacker path has been detected and killed, all resources owned by the path have been reclaimed.

4.4.4 Remarks

Note that many alternative policies are possible and easily enforced in Escort. For example, the passive path that

fields requests for new TCP connections can be given a limited share of the CPU, meaning that existing active paths are allowed to run in preference to starting new paths (creating new TCP connections). Similarly, clients that have previously violated some resource bound—e.g., the CGI attackers in our example—can be identified and their future connection request packets demultiplexed to a different distinct passive path with a very small resource allocation (or a very low priority). The possibility of IP spoofing, the presence of firewalls, and other aspects may also impact the policy that one chooses to implement. While we believe any such policy can be implemented in Escort, it is not clear that any single policy serves as a silver bullet for all possible denial of service attacks.

5 Related Work

Like Scout, Nemesis [14, 20] avoids cross talk by isolating data streams. It does not, however, take the additional step of accounting for all resource usage in a way that can be used to detect denial of service attacks. It also does not avoid cross talk when a data stream spans multiple protection domains. Escort's linkage and IPC model are also similar to Nemesis', as well as to other single address space operating systems [17, 11, 6].

Whereas Escort and Nemesis extend the operating system by moving functionality from the kernel to user space, Spin [4] and Vino [9] extend the OS by moving functionality into the kernel. However, all four systems face similar challenges. For example, [23] describes how transactions can be used in Vino to protect against misbehaving kernel extensions. The problem with this approach is that any single user of a kernel extension can consume all the extension's resources, even those allocated by other users. As a consequence, all the users of an extension have to trust each other.

Rushby [21] describes the security advantages of modeling a secure system after a distributed system. He argues that organizing an operating system in isolated protection domains which can only communicate via predefined channels as represented in our module graph makes arguing about and achieving high levels of security easier. We extend this idea by providing global QoS guarantees in the form of paths, and therefore enable such a system to deal with denial of service attacks.

LRPC [5] and migrating threads [10] are similar to Escort's thread model. Without the path abstraction, however, a migrating thread can be stopped only by destroying all the protection domains it crosses. This makes it substantially more difficult to defend against denial of service attacks.

6 Conclusions

This paper describes the Escort security architecture that we have implemented in the Scout operating system. Escort is novel in that it supports both end-to-end resource accounting (thereby protecting the system against denial of service attacks) and multiple hardware-enforced protection domains (thereby allowing untrusted modules to be isolated from each other).

We have used Escort to build a secure web server. Experiments with the server show that the accounting mechanism is highly accurate (accounting for virtually 100% of the cycles used to respond to HTTP requests), but imposes a relatively small overhead on the system (on the order of 8%). Enabling protection domains slows the system down by a factor of over four in the worst case measured. In practice, we expect the slowdown to be much less than a factor of two.

Finally, we demonstrate how Escort can be used to implement different denial of service policies. We measure three example policies and demonstrate that it is possible to detect and remove offending clients, while at the same time delivering quality-of-service guarantees to other clients. Although defining effective policies for various attacks is beyond the scope of this paper, we believe Escort provides the necessary mechanisms for implementing such policies.

Acknowledgments

We would like to thank the other members of the Scout group, particularly, Brady Montz and Andy Bavier. Thanks also to Inge Pudell-Spatscheck for editorial support, as to the reviewers, especially our shepherd, Paul Leach. This work was supported in part by DARPA Contract DABT63-95-C-0075 and NSF grant NCR-9204393.

References

[1] R. Atkinson. *RFC 1825 - Security Architecture for the Internet Protocol.* IETF, August 1995.

[2] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghigha. A domain and type enforcement UNIX prototype. In *Proceedings of the fifth USENIX UNIX Security Symposium: June 5–7, 1995, Salt Lake City, Utah, USA*, pages 127–140, Berkeley, CA, USA, June 1995. USENIX.

[3] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115–123, 1994.

[4] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the spin operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–284.

[5] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure calls. *ACM Transactions on Computer Systems, TOCS*, 8(1):37–55, Feb. 1990.

[6] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 80–85, 1993.

[7] T. Dierks and C. Allen. *Internet Draft: The TLS Protocol Version 1.0.* IETF, November 1997.

[8] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 189–202, New York, NY, USA, Dec. 1993. ACM Press.

[9] Y. Endo, J. Gwertzman, M. Seltzer, C. Small, K. A. Smith, and D. Tang. VINO: The 1994 fall harvest. Technical Report TR-34-94, Harvard University, Cambridge, MA, 1994.

[10] B. Ford and J. Lepreau. Evolving mach 3.0 to a migrating thread model. In *Proceedings of the Winter 1994 USENIX Technical Conference and Exhibition*, pages 97–114, Jan. 1994.

[11] G. Heiser, K. Elphinstone, S. Russell, and J. Vochteloo. Mungi: A distributed single address-space operating system. Technical Report SCS&E Report 9314, University of New South Wales, Australia, Nov. 1993.

[12] D. Mosberger. Message library design notes. Technical Report TR97-19, The Department of Computer Science, University of Arizona, Tuesday, Nov. 25 1997.

[13] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 153–167, Berkeley, CA, USA, Oct. 1996. USENIX.

[14] S. J. Mullender, I. M. Leslie, and D. McAuley. Operating system support for distributed multimedia. In *Proceedings of the Summer 1994 USENIX Conference: June 6–10, 1994, Boston, Massachusetts, USA*, pages 209–219, Berkeley, CA, USA, Summer 1994. USENIX.

[15] G. C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 15–17 Jan. 1997.

[16] P. G. Neumann. Architectures and formal representations for secure systems. Technical Report SRI-CSL-96-05, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1996.

[17] D. Probert and J. Bruno. Building fundamentally extensible application-specific operating system in SPACE. Technical Report TR95-06, Computer Science Department, University of California Santa Barbara, Oakland, CA, May 1995.

[18] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[19] T. Roscoe. Linkage in the Nemesis single address space operating system. *Operating Systems Review*, 28(4):48–55, Oct. 1994.

[20] T. Roscoe. *The Structure of a Multi-Service Operating System.* PhD thesis, University of Cambridge Computer Laboratory, April 1995.

[21] J. Rushby. The design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating System Principles, Asilomar CA*, pages 12–21, 1981.

[22] C. L. Schuba, I. Krsul, M. Kuhn, E. Spafford, A. Sundaram, and D. Zamboni. Analysis of denial of service attacks on TCP. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy, May 4-7, 1997. Oakland, California*, pages 208–223, Los Alamitos, CA, USA, May 1997. IEEE Computer Society Press.

[23] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 213–227, Berkeley, CA, USA, Oct. 1996. USENIX.

[24] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of 14th ACM SOSP*, pages 175–188, Asheville, NC, Dec. 1993.

Self-Paging in the Nemesis Operating System

Steven M. Hand
University of Cambridge Computer Laboratory
New Museums Site, Pembroke St.,
Cambridge CB2 3QG, ENGLAND
Steven.Hand@cl.cam.ac.uk

Abstract

In contemporary operating systems, continuous media (CM) applications are sensitive to the behaviour of other tasks in the system. This is due to contention in the kernel (or in servers) between these applications. To properly support CM tasks, we require "Quality of Service Firewalling" between different applications.

This paper presents a memory management system supporting Quality of Service (QoS) within the *Nemesis* operating system. It combines application-level paging techniques with isolation, exposure and responsibility in a manner we call *self-paging*. This enables rich virtual memory usage alongside (or even within) continuous media applications.

1 Introduction

Researchers now recognise the importance of providing support for continuous media applications within operating systems. This is evinced by the *Nemesis* [1, 2, 3] and *Rialto* [4, 5, 6] operating systems and, more recently, work on the *Scout* [7] operating system and the SMART scheduler [8]. Meanwhile there has been continued interest in the area of memory management, with a particular focus on *extensibility* [9, 10, 11].

While this work is valid, it is insufficient:

- Work on continuous media support in operating systems tends to focus on CPU scheduling only.
  The area of memory management is either totally ignored (Scout, SMART) or considered in practice to be a protection mechanism (Rialto). In fact, the implementation of the Rialto virtual memory system described in [12] explicitly excludes paging since it "*introduces unpredictable latencies*".
- Work on memory management does not support (or try to support) any concept of Quality of Service.
  While support for extensibility is a laudable goal, the behaviour of user-level pagers or application-provided code is hardly any more predictable or isolated than kernel-level implementations. The "unpredictable latencies" remain.

This paper presents a scheme whereby each application is responsible for all of its own paging (and other virtual memory activities). By providing applications with guarantees for physical memory and disk bandwidth, it is possible to isolate time-sensitive applications from the behaviour of others.

2 Quality of Service in Operating Systems

In recent years, the application mix on general purpose computers has shifted to include "multimedia" applications. Of particular interest are *continuous media* (CM) applications — those which handle audio and/or video — since the presentation (or processing) of the information must be done in a timely manner. Common difficulties encountered include ensuring low latency (especially for real-time data) and minimising *jitter* (viz. the variance in delay).

Clearly not all of today's applications have these temporal constraints. More traditional tasks such as formatting a document, compiling a program, or sending e-mail are unlikely to be banished by emerging continuous media applications. Hence there is a requirement for *multi-service* operating systems which can support both types of application simultaneously.

Unfortunately, most current operating systems conspicuously fail to support this mix of CM and non-CM applications:

- CPU scheduling is usually implemented via some form of priority scheme, which specifies *who* but not *when* or *how much*. This is unfortunate since many continuous media applications do not require a large fraction of the CPU resource (i.e. they are not necessarily more *important* than other applications), but

they do need to be scheduled in a *timely* fashion.

- Other resources on the data path, such as the disk or network, are generally not explicitly scheduled at all. Instead, the proportion of each resource granted to an application results from a complex set of unpredictable interactions between the kernel (or user-level servers) and the CPU scheduler.
- The OS performs a large number of (potentially) time-critical tasks on behalf of applications. The performance of any particular application is hence heavily dependent on the execution of other supposedly "independent" applications. A greedy, buggy or even pathological application can effect the degradation of all other tasks in the system.

This means that while most systems can support CM applications in the case of resource over-provisioning, they tend to exhibit poor behaviour when contention is introduced.

A number of operating systems researchers are now attempting to provide support for continuous media applications. The Rialto system, for example, hopes to provide modular real-time resource management [4] by means of arbitrarily composable *resource interfaces* collectively managed by a *resource planner*. A novel real-time CPU scheduler has been presented in [5, 6], while an implementation of a simple virtual memory system for set-top boxes is described in [12].

The Scout operating system uses the *path* abstraction to ensure that continuous media streams can be processed in a timely manner. It is motivated by multimedia network streams, and as such targets itself at sources (media servers) and sinks (set-top boxes) of such traffic. Like Rialto, the area of virtual memory management is not considered a high-priority; instead there is a rudimentary memory management system which focuses upon buffer management and does not support paging.

Most other research addresses the problem of Quality of Service within a specific domain only. This has lead to the recent interest in soft real-time scheduling [13, 8, 14, 15] of the CPU and other resources. The work has yet to be widely applied to multiple resources, or to the area of memory management.

3 Extensible Memory Management

Memory management systems have a not undeserved reputation for being complex. One successful method of simplifying the implementation has been the μ-kernel approach: move some or all of the memory management system out of the kernel into "user-space". This mechanism, pioneered by work on Mach [16] is still prevalent in many modern μ-kernels such as V++ [17], Spring [18] and L4 [19].

Even operating systems which eschew the μ-kernel approach still view the extensibility of the memory management system as important:

- the *SPIN* operating system provides for user-level extension of the memory management code via the registration of an event handler for memory management. events [10].
- the VINO operating system [20, pp 1–6] enables applications to override some or all operations within *MemoryResource* objects, to specialise behaviour.
- the V++ Cache Kernel allows "application kernels" to cache address-space objects and to subsequently handle memory faults on these [21].
- the Aegis experimental exokernel enables "library operating systems" to provide their own page-table structures and TLB miss handlers [9].

This is not surprising. Many tasks are ill-served by default operating system abstractions and policies, including database management (DBMS) [22], garbage collection [23] and multi-media applications [24]. Furthermore, certain optimisations are possible when application-specific knowledge can be brought to bear, including improved page replacement and prefetching [17], better buffer cache management [25], and light-weight signal handling [26]. All of these may be realised by providing user-level control over some of the virtual memory system.

Unfortunately, none of the above-mentioned operating systems provide QoS in their memory management:

- No Isolation: applications which fault repeatedly will still degrade the overall system performance. In particular, they will adversely affect the operation of other applications.
  In μ-kernel systems, for example, a single external pager may be shared among an arbitrary number of processes, but there is no scheduling regarding fault resolution. This indirect contention has been referred to as *QoS crosstalk* [2]. Other extensible systems allow the application to specify, for example, the page replacement *policy*, but similarly fail to arbitrate between multiple faulting applications.
- Insufficient Exposure: most of the above operating systems[1] abstract away from the underlying hardware; memory faults are presented as some abstract form of exception and memory translation as an array of virtual to physical mappings.
  Actual hardware features such as multiple TLB page sizes, additional protection bits, address space numbers, physical address encoding, or cache behaviour tend to be lost as a result of this abstraction.

[1] A notable exception is the Aegis exokernel, which endeavours to expose as much as possible to the application.

- No Responsibility: while the many of the above operating systems *allow* applications some form of "extensibility" for performance or other reasons, they do not by any means *enforce* its use. Indeed, they provide a "default" or "system" pager to deal with satisfying faults in the general case. The use of this means that most applications fail to pay for their own faults; instead the system pager spends its time and resources processing them.

What is required is a system whereby applications benefit from the ability to control their own memory management, but do not gain at the expense of others.

4 Nemesis

The Nemesis operating system has been designed and implemented at the University of Cambridge Computer Laboratory in recent years. Nemesis is a *multi-service* operating system — that is, it strives to support a mix of conventional and time-sensitive applications. One important problem it addresses is that of preventing *QoS crosstalk*. This can occur when the operating system kernel (or a shared server) performs a significant amount of work on behalf of a number of applications. For example, an application which plays a motion-JPEG video from disk should not be adversely affected by a compilation started in the background.

One key way in which Nemesis supports this isolation is by having applications execute as many of their own tasks as possible. This is achieved by placing much traditional operating system functionality into user-space modules, resulting in a *vertically integrated* system (as shown in Figure 1). This vertical structure is similar to that of the Cache Kernel [21] and the Aegis Exokernel [27], although the motivation is different.

The user-space part of the operating system is comprised of a number of distinct *modules*, each of which exports one or more strongly-typed *interfaces*. An interface definition language called *MIDDL* is used to specify the types, exceptions and procedures of an interface, and a run-time typesystem allows the narrowing of types and the marshaling of parameters for non-local procedure invocations.

A name-space scheme (based on Plan-9 contexts) allows implementations of interfaces to be published and applications to pick and choose between them. This may be termed "plug and play extensibility"; we note that it is implemented *above* the protection boundary.

Given that applications are responsible for executing traditional operating system functions themselves, the must be sufficiently empowered to perform them. Nemesis handles this by providing explicit low-level resource guarantees or reservations to applications. This is not limited simply to the CPU: all resources — including disks [14], network

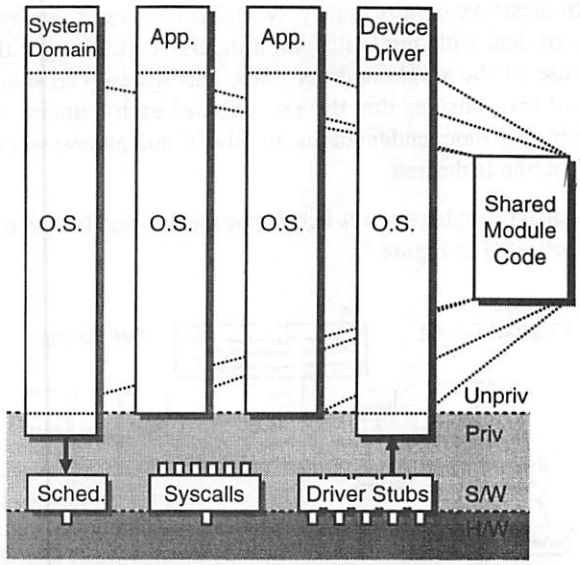


Figure 1: Vertical Structuring in Nemesis

interfaces [28] and physical memory — are treated in the same way. Hence any given application has a set of guarantees for all the resources it requires. Other applications cannot interfere.

5 Self-Paging

Self-paging provides a simple solution to memory system crosstalk: *require every application to deal with all its own memory faults using its own concrete resources*. All paging operations are removed from the kernel; instead the kernel is simply responsible for dispatching fault notifications.

More specifically, self-paging involves three principles:

1. *Control*: resource access is multiplexed in both space and time. Resources are guaranteed over medium-term time-scales.
2. *Power*: interfaces are sufficiently expressive to allow applications the flexibility they require. High-level abstractions are not imposed.
3. *Responsibility*: each application is directly responsible for carrying out its own virtual memory operations. The system does not provide a "safety net".

The idea of performing virtual memory tasks at application-level may at first sound similar to the ideas pioneered in Mach [16] and subsequently used in μ-kernel systems such as L4 [19]. However while μ-kernel systems *allow* the use of one or more external (i.e. non-kernel) pagers in order to provide extensibility and to simplify the kernel, several applications will typically still share an external pager, and hence the problem of QoS crosstalk remains.

In Nemesis we *require* that every application is *self-paging*. It *must* deal with any faults that it incurs. This, along with the use of the single address space and widespread sharing of text, ensures that the execution of each domain[2] is completely independent to that of other domains save when interaction is desired.

The difference between μ-kernel approaches and Nemesis' is illustrated in Figure 2.

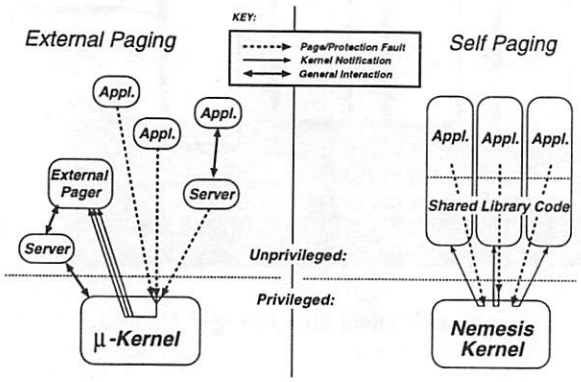


Figure 2: External Paging versus Self Paging

The left-hand side on the figure shows the μ-kernel approach, with an external pager. Three applications are shown, with two of them causing page faults (or, more generally, memory faults). The third application has a server performing work on its behalf, and this server is also causing a memory fault. The kernel notifies the external pager and it must then deal with the three faults.

This causes two problems:

1. Firstly, the process which caused the fault does not use any of its own resources (in particular, CPU time) in order to satisfy the fault. There is no sensible way in which the external pager (or the μ-kernel itself) can account for this. A process which faults repeatedly thus degrades the overall system performance but bears only a fraction of the cost.

2. Secondly, multiplexing happens in the server — i.e. the external pager needs some way to decide how to 'schedule' the handling of the faults. However it will generally not be aware of any absolute (or even relative) timeliness constraints on the faulting clients. A first-come first-served approach is probably the best it can do.

On the right-hand side we once again have three applications, but no servers. Each application is causing a memory fault of some sort, which is trapped by the kernel. However rather than sending a notification to some external pager, the kernel simply notifies the faulting domain. Each domain will itself be responsible for handling the fault. Furthermore, the latency with which the fault will be resolved (assuming it is resolvable) is dependent on the guarantees held by that domain.

Rather closer to the self-paging ideal are "vertically structured" systems such as the Aegis & Xok exokernels [9, 29]. Like Nemesis, these systems dispatch memory faults to user-space and expect unprivileged library operating system code to handle them. In addition, exokernels expose sufficient low-level detail to allow applications access to hardware-specific resources.

However exokernels do not fully cope with the aspect of *control*: resources are multiplexed in space (i.e. there is protection), but not in time. For example, the Xok exokernel allows library filing systems to download untrusted metadata translation functions. Using these in a novel way, the exokernel can protect disk blocks without understanding file systems [29]. Yet there is no consideration given to partitioning access in terms of *time*: library filing systems are not guaranteed a proportion of disk bandwidth.

A second problem arises with crosstalk within the exokernel itself. Various device drivers coexist within the kernel execution environment and hence an application (or library operating system) which is paging heavily will impact others who are using orthogonal resources such as the network. This problem is most readily averted by pushing device driver functionality outside the kernel, as is done with μ-kernel architectures.

6 System Design

A general overview of the virtual memory architecture is shown in Figure 3. This is necessarily simplified, but does illustrate the basic abstractions and their relationships.

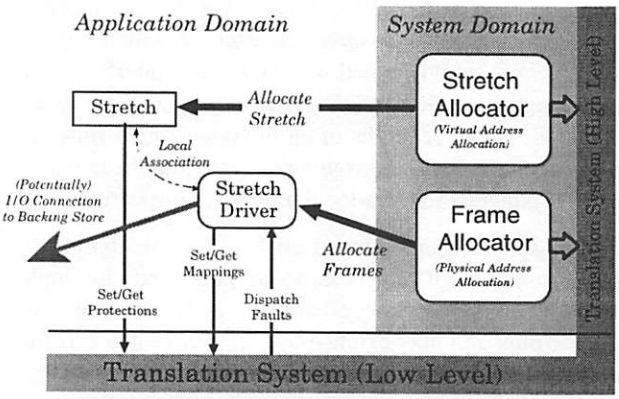


Figure 3: VM System Architecture

The basic virtual memory abstractions shown are the *stretch* and the *stretch driver*. A stretch merely represents a range of virtual addresses with a certain accessibility. It

[2] A *domain* in Nemesis is the analog of a process or task.

does not own — nor is it guaranteed — any physical resources. A stretch driver is responsible for providing the backing for the stretch; more generally, a stretch driver is responsible for dealing with any faults on a stretch. Hence it is only via its association with a stretch driver that it becomes possible to talk meaningfully about the "contents" of a stretch. Stretch drivers are unprivileged, application-level objects, instantiated by user-level creator modules and making use only of the resources owned by the application.

The stretch driver is shown as potentially having a connection to a backing store. This is necessarily vague: there are many different sorts of stretch driver, some of which do not deal with non-volatile storage at all. There are also potentially many different kinds of backing store. The most important of these is the *User-Safe Backing Store* (USBS). This draws on the work described in [14] to provide per-application guarantees on paging bandwidth, along with isolation between paging and file-system clients.

Allocation is performed in a centralised way by the system domain, for both virtual and physical memory resources. The high-level part of the translation system is also in the system domain: this is machine-dependent code responsible for the construction of page tables, and the setting up of "NULL" mappings for freshly allocated virtual addresses. These mappings are used to hold the initial protection information, and by default are set up to cause a page fault on the first access. Placing this functionality within the system domain means that the low-level translation system does not need to be concerned with the allocation of page-table memory. It also allows protection faults, page faults and "unallocated address" faults to be distinguished and dispatched to the faulting application.

Memory protection operations are carried out by the application through the stretch interface. This talks directly to the low-level translation system via simple system calls; it is not necessary to communicate with the system domain. Protection can be carried out in this way due to the protection model chosen which includes explicit rights for "change permissions" operations. A light-weight validation process checks if the caller is authorised to perform an operation.

The following subsections explain relevant parts of this architecture in more detail.

6.1 Virtual Address Allocation

Any domain may request a stretch from a stretch allocator, specifying the desired size and (optionally) a starting address and attributes. Should the request be successful, a new stretch will be created and returned to the caller. The caller is now the *owner* of the stretch. The starting address and length of the returned stretch may then be queried; these will always be a multiple of the machine's page size[3].

Protection is carried out at stretch granularity — every *protection domain* provides a mapping from the set of valid stretches to a subset of { *read*, *write*, *execute*, *meta* }. A domain which holds the *meta* right is authorised to modify protections and mappings on the relevant stretch.

When allocated, a stretch need not in general be backed by physical resources. Before the virtual address may be referred to the stretch must be associated with a *stretch driver* — we say that a stretch must be *bound* to a stretch driver. The stretch driver is the object responsible for providing any backing (physical memory, disk space, etc.) for the stretch. Stretch drivers are covered in Section 6.6.

6.2 Physical Memory Management

As with virtual memory, the allocation of physical memory is performed centrally, in this case by the *frames allocator*. The frames allocator allows fine-grained control over the allocation of physical memory, including I/O space if appropriate. A domain may request specific physical frames, or frames within a "special" region[4]. This allows an application with platform knowledge to make use of page colouring [30], or to take advantages of superpage TLB mappings, etc. A default allocation policy is also provided for domains with no special requirements.

Unlike virtual memory, physical memory is generally a scarce resource. General purpose operating systems tend to deal with contention for physical memory by performing *system-wide load balancing*. The operating system attempts to (dynamically) share physical memory between competing processes. Frames are *revoked* from one process and granted to another. The main motivation is global system performance, although some systems may consider other factors (such as the estimated working set size or process class).

Since in Nemesis we strive to devolve control to applications, we use an alternative scheme. Each application has a contract with the frames allocator for a certain number of *guaranteed* physical frames. These are immune from revocation in the short term (on the order of tens of seconds). In addition to these, an application may have some number of *optimistic* frames, which may be revoked at much shorter notice. This distinction only applies to frames of main memory, not to regions of I/O space.

When a domain is created, the frames allocator is requested to admit it as a client with a service contract $\{g, x\}$. This represents a pair of quotas for guaranteed and optimistic

[3] Where multiple page sizes are supported, "page size" refers to the size of the smallest page.

[4] Such as DMA-accessible memory on certain architectures.

frames respectively. Admission control is based on the requested guarantee g — the sum of all guaranteed frames contracted by the allocator must be less than the total amount of main memory. This is to ensure that the guarantees of all clients can be met simultaneously.

The frames allocator maintains the tuple $\{n, g, x\}$ for each client domain, where n is the number of physical frames allocated so far. As long as $g > n$, a request for a single physical frame is guaranteed to succeed[5]. If $g \leq n < x$ and there is available memory, frames will be *optimistically* allocated to the caller.

The allocation of optimistic frames improves global performance by allowing applications to use the available memory when it is not otherwise required. If, however, a domain wishes to use some more of the frames guaranteed to it, it may be necessary to *revoke* some optimistically allocated frames from another domain. In this case, the frames allocator chooses a candidate application[6], but the selection of the frames to release (and potentially write to the backing store) is under the control of the application.

By convention, each application maintains a *frame stack*. This is a system-allocated data structure which is writable by the application domain. It contains a list of physical frame numbers (PFNs) owned by that application ordered by 'importance' — the top of the stack holds the PFN of the frame which that domain is most prepared to have revoked.

This allows revocation to be performed *transparently* in the case that the candidate application has *unused* frames at the top of its stack. In this case, the frames allocator can simply reclaim these frames and update the application's frame stack. Transparent revocation is illustrated on the left-hand side of Figure 4.

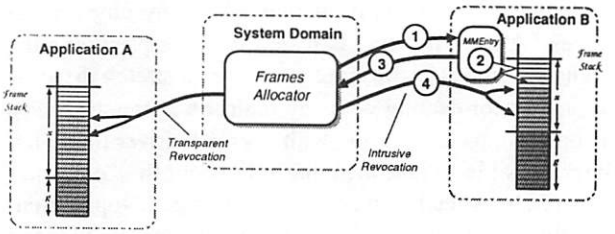
If there are no unused frames available, *intrusive* revocation is required. In this case, the frames allocator sends a revocation notification to the application requesting that it release k frames by time T. The application then must arrange for the top k frames of its frame stack to contain unmapped frames. This can require that it first clean some dirty pages; for this reason, T may be relatively far in the future (e.g. 100ms).

After the application has completed the necessary operations, it informs the frames allocator that the top k frames may now be reclaimed from its stack. If these are not all unused, or if the application fails to reply by time T, the domain is killed and all of its frames reclaimed. This protocol is illustrated on the right-hand side of Figure 4.

Notice that since the frames allocator *always* revokes from the top of an application's frame stack, it makes sense for the application to maintain its preferred revocation order.

[5]Due to fragmentation, a single request for up to $(g - n)$ frames may or may not succeed.

[6]i.e. one which currently has optimistically allocated frames.



① The frames allocator sends a revocation notification to Application B.
② Application B fields this notification, and arranges for the top k frames on the stack to be unused.
③ Application B replies that all is now ready.
④ The frames allocator reclaims the top k frames from the stack.

Figure 4: Revocation of Physical Memory

The frame stack also provides a useful place for stretch drivers to store local information about mappings, and enables the internal revocation interface to be simpler.

Due to the need for relatively large timeouts, a client domain requesting some more guaranteed frames may have to wait for a non-trivial amount of time before its request succeeds. Hence time-sensitive applications generally request *all* their guaranteed frames during initialisation and do not use optimistically allocated frames at all. This is not mandated, however: use of physical memory is considered orthogonal to use of other resources. The only requirement is that any domain which uses optimistically allocated frames should be able to handle a revocation notification.

6.3 Translation System

The translation system deals with inserting, retrieving or deleting mappings between virtual and physical addresses. As such it may be considered an interface to a table of information held about these mappings; the actual mapping will typically be performed as necessary by whatever memory management hardware or software is present.

The translation system is divided into two parts: a high-level management module, and the low-level trap handlers and system calls. The high-level part is private to the system domain, and handles the following:

- Bootstrapping the 'MMU' (in hardware or software), and setting up initial mappings.
- Adding, modifying or deleting ranges of virtual addresses, and performing the associated page table management.
- Creating and deleting protection domains.

- Initialising and partly maintaining the *RamTab*; this is a simple data structure maintaining information about the current use of frames of main memory.

The high-level translation system is used by both the stretch allocator and the frames allocator. The former uses it to setup initial entries in the page table for stretches it has created, or to remove such entries when a stretch is destroyed. These entries contain protection information but are by default *invalid*: i.e. addresses within the range will cause a page fault if accessed. The frames allocator, on the other hand, uses the *RamTab* to record the owner and logical frame width of allocated frames of main memory.

Recall that each domain is expected to deal with mapping its own stretches. The low-level translation system provides direct support for this to happen efficiently and securely. It does this via the following three operations:

1. `map(va, pa, attr)` : arrange that the virtual address `va` maps onto the physical address `pa` with the (machine-dependent) PTE attributes `attr`.
2. `unmap(va)` : remove the mapping of the virtual address `va`. Any further access to the address should cause some form of memory fault.
3. `trans(va)` → `(pa, attr)` : retrieve the current mapping of the virtual address `va`, if any.

Either mapping or unmapping a virtual address `va` requires that the calling domain is executing in a protection domain which holds a *meta* right for the stretch containing `va`. A consequence of this is that it is not possible to map a virtual address which is not part of some stretch[7].

It is also necessary that the frame which is being used for mapping (or which is being unmapped) is validated. This involves ensuring that the calling domain owns the frame, and that the frame is not currently mapped or nailed. These conditions are checked by using the *RamTab*, which is a simple enough structure to be used by low-level code.

6.4 Fault Dispatching

Apart from TLB misses which are handled by the low-level translation system, all other faults are dispatched directly to the faulting application in order to prevent QoS crosstalk. To prevent the need to block in the kernel for a user-level entity, the kernel-part of fault handling is complete once the dispatch has occurred. The application must perform any additional operations, including the resumption (or termination) of the faulting thread.

The actual dispatch is achieved by using an *event channel*. Events are an extremely lightweight primitive provided by the kernel — an event "transmission" involves a few sanity checks followed by the increment of a 64-bit value. A full description of the Nemesis event mechanism is given in [2].

On a memory fault, then, the kernel saves the current context in the domain's *activation context* and sends an event to the faulting domain. At some point in the future the domain will be selected for activation and can then deal with the fault. Sufficient information (e.g. faulting address, cause, etc.) is made available to the application to facilitate this. Once the fault has been resolved, the application can resume execution from the saved activation context.

6.5 Application-Level Fault Handling

At some point after an application has caused a memory fault, it will be *activated* by the scheduler. The application then needs to handle all the events it has received since it was last activated. This is achieved by invoking a *notification handler* for each endpoint containing a new value; if there is no notification handler registered for a given endpoint, no action is taken. Following this the user-level thread scheduler (ULTS) is entered which will select a thread to run.

Up until the point where a thread is run, the application is said to be running within an *activation handler*. This is a limited execution environment where further activations are disallowed. One important restriction is that inter-domain communication (IDC) is not possible within an activation handler. Hence if the handling of an event requires communication with another domain, the relevant notification handler simply unblocks a worker thread. When this is scheduled, it will carry out the rest of the operations required to handle the event.

The combination of notification handler and worker threads is called an *entry* (after ANSAware/RT [31]). Entries encapsulate a scheduling policy on event handling, and may be used for a variety of IDC services. An entry called the *MMEntry* is used to handle memory management events.

The notification handler of the *MMEntry* is attached to the endpoint used by the kernel for fault dispatching. Hence it gets an upcall every time the domain causes a memory fault. It is also entered when the frames allocator performs a revocation notification (as described in Section 6.2). The 'top' part of the *MMEntry* consists of one or more worker threads which can be unblocked by the notification handler.

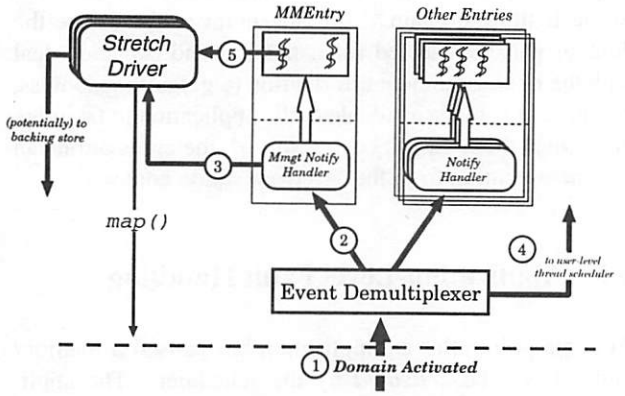
The *MMEntry* does not directly handle memory faults or revocation requests: rather it coordinates the set of stretch drivers used by the domain. It does this in one of two ways:

- If handling a memory fault, it uses the faulting stretch to lookup the stretch driver bound to that stretch and then invokes it.

[7] Bootstrapping code clearly does this, but it uses the high-level translation system and not this interface.

- If handling a revocation notification, it cycles through each stretch driver requesting that it relinquish frames until enough have been freed.

Figure 5 illustrates this in the case of a page fault.



① The domain receives an event. At some point, the kernel decides to schedule it, and it is *activated*. It is informed that the reason for the activation was the receipt of an event.
② The user-level event demultiplexer notifies interested parties of any events which have been received on their end-point(s).
③ The memory fault notification handler demultiplexes the stretch to the stretch driver, and invokes this in an initial attempt to satisfy the fault.
④ If the attempt fails, the handler blocks the faulting thread, unblocks a worker thread, and returns. After all events have been handled, the user-level thread scheduler is entered.
⑤ The worker thread in the memory management entry is scheduled and once more invokes the stretch driver to map the fault, which may potentially involve communication with another domain.

Figure 5: Memory Management Event Handling

Note that the initial attempt to resolve the fault (arrow labelled ③) is merely a "fast path" optimisation. If it succeeds, the faulting thread will be able to continue once the ULTS is entered. On the other hand, if the initial attempt fails, the *MMEntry* must block the faulting thead pending the resolution of the fault by a worker thread.

6.6 Stretch Drivers

As has been described, the actual resolution of a fault is the province of a *stretch driver*. A stretch driver is something which provides physical resources to back the virtual addresses of the stretches it is responsible for. Stretch drivers acquire and manage their own physical frames, and are responsible for setting up virtual to physical mappings by invoking the translation system.

The current implementation includes three stretch drivers which may be used to handle faults. The simplest is the *nailed* stretch driver; this provides physical frames to back a stretch at bind time, and hence never deals with page faults. The second is the *physical* stretch driver. This provides no backing frames for any virtual addresses within a stretch initially. The first authorised attempt to access any virtual address within a stretch will cause a page fault which is dispatched in the manner described in Section 6.4. The physical stretch driver is invoked from within the *notification handler*: this is a limited execution environment where certain operations may occur but others cannot. Most importantly, one cannot perform any inter-domain communication (IDC) within a notification handler.

When the stretch driver is invoked, the following occurs:

- After performing basic sanity checks, the stretch driver looks for an unused (i.e. unmapped) frame. If this fails, it cannot proceed further now — but may be able to request more physical frames when activations are on. Hence it returns `Retry`.

- Otherwise, it can proceed now. In this case, the stretch driver sets up the new mapping with a call to `map(va,pa,attr)`, and returns `Success`.

In the case where `Retry` is returned, a memory management entry worker thread will invoke the physical stretch driver for a second time once activations are on. In this case, IDC operations are possible, and hence the stretch driver may attempt to gain additional physical frames by invoking the frames allocator. If this succeeds, the stretch driver sets up a mapping from the faulting virtual address to a newly allocated physical frame. Otherwise the stretch driver returns `Failure`.

The third stretch driver implemented is the *paged* stretch driver. This may be considered an extension of the physical stretch driver; indeed, the bulk of its operation is precisely the same as that described above. However the paged stretch driver also has a binding to the USBS and hence may swap pages in and out to disk. It keeps track of swap space as a bitmap of *bloks* — a blok is a contiguous set of disk blocks which is a multiple of the size of a page. A (singly) linked list of bitmap structures is maintained, and bloks are allocated first fit — a hint pointer is maintained to the earliest structure which is known to have free bloks.

Currently we implement a fairly pure demand paged scheme — when a page fault occurs which cannot be satisfied from the pool of free frames, disk activity of some form will ensue. Clearly this can be improved; however it will suffice for the demonstration of "Quality of Service Firewalling" in Section 7.2.

6.7 User-Safe Backing Store

The user-safe backing store (USBS) is comprised of two parts: the swap filesystem (SFS) and the user-safe disk (USD) [32]. The SFS is responsible for *control* operations such as allocation of an extent (a contiguous range of blocks) for use as a swap file, and the negotiation of Quality of Service parameters to the USD, which is responsible for scheduling *data* operations. This is illustrated in Figure 6.

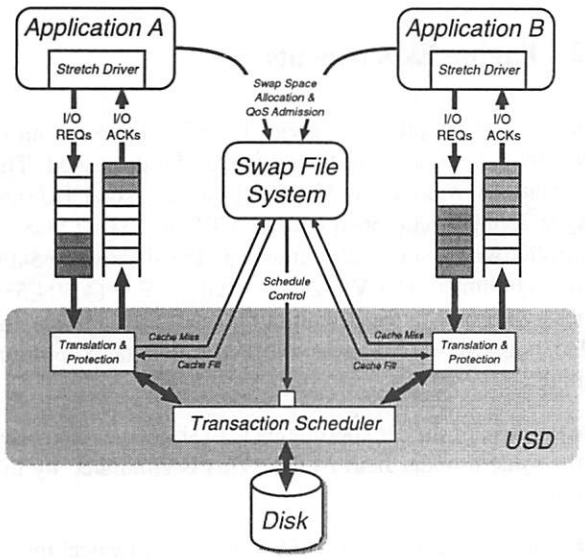


Figure 6: The User-Safe Backing Store

Clients communicate with the USD via a FIFO buffering scheme called *IO channels*; these are similar in operation to the 'rbufs' scheme described in [33].

The type of QoS specification used by the USD is in the form (p, s, x, l) where p is the *period* and s the *slice*; both of these are typically of the order of tens of milliseconds. Such a guarantee represents that the relevant client will be allowed to perform disk transactions totalling at most s ms within every p ms period. The x flag determines whether or not the client is eligible for any slack time which arises in the schedule — for the purposes of this paper it will always be *False*, and so may be ignored.

The actual scheduling is based on the *Atropos* algorithm [2]: this is a based on the earliest-deadline first (EDF) algorithm [34], although the deadlines are implicit, and there is support for *optimistic* scheduling.

Each client is periodically allocated s ms and a deadline of *now* + p ms, and placed on a *runnable* queue. A thread in the USD domain is awoken whenever there are pending requests and, if there is work to be done for multiple clients, chooses the one with the earliest deadline and performs a single transaction.

Once the transaction completes, the time taken is computed and deducted from that client's remaining time. If the remaining time is ≤ 0, the client is moved onto a *wait* queue; once its deadline is reached, it will receive a new allocation and be returned to the runnable queue. Until that time, however, it cannot perform any more operations.

Note that this algorithm will tend to perform requests from a single client consecutively. This is a very desirable property since it minimises the impact of clients upon each other — the first transaction after a "context switch" to a new client will often incur a considerable seek/rotation penalty over which it has no control. However this cost can be amortised over the number of transactions which the client subsequently carries out, and hence has a smaller impact on its overall performance.

Unfortunately, many clients (and most particularly clients using the USD as a swap device) cannot pipeline a large number of transactions since they do not know in advance to/from where they will wish to write/read. Early versions of the USD scheduler suffered from this so-called "short-block" problem: if the client with the earliest deadline has (instantaneously) no further work to be done, the USD scheduler would mark it idle, and ignore it until its next periodic allocation.

To avoid this problem, the idea of "laxity" is used, as given by the l parameter of the tuple mentioned above. This is a time value (typically a small number of milliseconds) for which a client should be allowed to remain on the runnable queue, even if it currently has no transactions pending. This does not break the scheduling algorithm since the additional time spent — the *lax time* — is accounted to the client just as if it were time spent performing disk transactions. Section 7.2 will show the beneficial impact of laxity in the case of paging operations.

7 Experiments

7.1 Micro-Benchmarks

In order to evaulutate the combination of low-level and application-level memory system functions, a set of micro-benchmarks based on those proposed in [23] were performed on Nemesis and compared with Digital OSF1 V4.0 on the same hardware (PC164) and basic page table structure (linear). The results are shown in Table 1.

The first benchmark shown is `dirty`. After [9] this measures the time to determine whether a page is dirty or not. On Nemesis this simply involves looking up a random page table entry and examining its 'dirty' bit[8]. We use a *linear* page table implementation (i.e. the main page table is an 8Gb array in the virtual address space with a secondary

[8] We implement 'dirty' and 'referenced' using the FOR/FOW bits; these are set by software and cleared by the PALCODE DFault routine.

OS	dirty	(un)prot1	(un)prot100
OSF1 V4.0	n/a	3.36	5.14
Nemesis	0.15	0.42 [0.40]	10.78 [0.30]
	trap	**appel1**	**appel2**
OSF1 V4.0	10.33	24.08	19.12
Nemesis	4.20	5.33	9.75†

†Non-standard — see main text.

Table 1: Comparative Micro-Benchmarks; the units are μs.

page table used to map it on "double faults") which provides efficient translation; an earlier implementation using *guarded* page tables was about three times slower.

The second benchmark measures the time taken to protect or unprotect a random page. Since our protection model requires that all pages of a stretch have the same access permissions, this amounts to measuring the time required to change the permissions on small stretches. There are two ways to achieve this under Nemesis: by modifying the page tables, or by modifying a *protection domain* — the times for this latter procedure are shown in square brackets.

The third benchmark measures the time taken to (un)protect a range of 100 pages. Nemesis does not have code optimised for the page table mechanism (e.g. it looks up each page in the range individually) and so it takes considerably longer than (un)protecting a single page. OSF1, by contrast, shows only a modest increase in time when protecting more than one page at a time. Nemesis does perform well when using the protection domain scheme.

This benchmark is repeated a number of times with the same range of pages and the average taken. Since on Nemesis the protection scheme detects idempotent changes, we alternately protect and unprotect the range; otherwise the operation takes an average of only 0.15μs. If OSF1 is benchmarked using the Nemesis semantics of alternate protections, the cost increases to ~75μs.

The `trap` benchmark measures the time to handle a page fault in user-space. On Nemesis this requires that the kernel send an event (<50ns), do a full context save (~750ns), and then activate the faulting domain (<200ns). Hence approximately 3μs are spent in the unoptimised user-level notification handlers, stretch drivers and thread-scheduler. This could clearly be improved.

The next benchmark, `appel1` (this is "prot1+trap+unprot" in [23]), measures the time taken to access a random protected page and, in the fault handler, to unprotect that page and protect another. This uses a standard (physical) stretch driver with the access violation fault type overridden by a custom fault-handler; a more efficient implementation would use a specialised stretch driver.

The final benchmark is `appel2`, which is called "protN+trap+unprot" in [23]. This measures the time taken to protect 100 contiguous pages, to access each in a random order and, in the fault handler, unprotect the relevant page. It is not possible to do this precisely on Nemesis due to the protection model — all pages of a stretch must have the same accessibility. Hence we unmap all pages rather than protecting them, and map than rather than unprotecting them. An alternative solution would have been use the Alpha FOW bit, but this is reserved in the current implementation.

7.2 Paging Experiments

A number of simple experiments have been carried out to illustrate the operation of the system so far described. The host platform was a Digital EB164 (with a 21164 CPU running at 266Mhz) equipped with a NCR53c810 Fast SCSI-2 controller with a single disk attached. The disk was a 5400 rpm Quantum (model VP3221), 2.1Gb in size (4,304,536 blocks with 512 bytes per block). Read caching was enabled, but write caching was disabled (the default configuration).

The purpose of these experiments is to show the behaviour of the system under heavy load. This is simulated by the following:

- Each application has a tiny amount of physical memory (16Kb, or 2 physical frames), but a reasonable amount of virtual memory (4Mb).
- A trivial amount of computation is performed per page — in the tests, each byte is read/written but no other substantial work is performed.
- No pre-paging is performed, despite the (artificially) predictable reference pattern.

A test application was written which created a paged stretch driver with 16Kb of physical memory and 16Mb of swap space, and then allocated a 4Mb stretch and bound it to the stretch driver. The application then proceeded to sequentially read every byte in the stretch, causing every page to be demand zeroed.

7.2.1 Paging In

The first experiment is designed to illustrate the overall performance and isolation achieved when multiple domains are paging in data from different parts of the same disk. The test application continues from the initialisation described above by writing to every byte in the stretch, and then forking a "watch thread". The main thread continues sequentially accessing every byte from the start of the 4Mb stretch, incrementing a counter for each byte 'processed' and looping around to the start when it reaches the top.

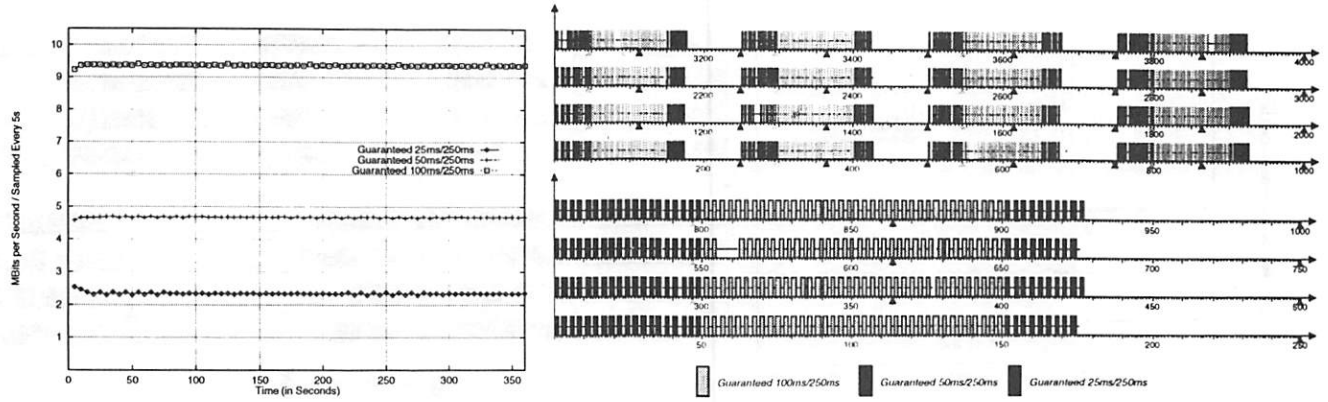


Figure 7: Paging In (*lhs* shows sustained bandwidth, *rhs* shows a USD Scheduler Trace)

The watch thread wakes up every 5 seconds and logs the number of bytes processed.

The experiment uses three applications: one is allocated 25ms per 250ms, the second allocated 50ms per 250ms, and the third allocated 100ms per 250ms — the same period is used in each case to make the results easier to understand. No domain is eligible for slack time, and all domains have a *laxity* value of 10ms. The resulting measured progress (in terms of Mbits/second) is plotted on the left hand side of Figure 7.

Observe that the ratio between the three domains is very close to 4 : 2 : 1, which is what one would expect if each domain were receiving all of its guaranteed time. In order to see what was happening in detail, a log was taken inside the USD scheduler which recorded, among other things:

- each time a given client domain was scheduled to perform a transaction,
- the amount of lax time each client was charged, and
- the period boundaries upon which a new allocation was granted.

The right hand side of Figure 7 shows these events on two different time-scales: the top plot shows a four second sample, while the bottom plot shows the first second of the sample in more detail. The darkest gray squares represent transactions from the application with the smallest guarantee (10%), while the lightest gray show those from the application with the highest (40%). The small arrows in each colour represent the point at which the relevant client received a new allocation.

Each filled box shows a transaction carried out by a given client — the width of the box represents the amount of time the transaction took. All transactions in the sample given take roughly the same time; this is most likely due to the fact that the sequential reads are working well with the cache.

The solid lines between the transactions (most visible in the detailed plot) illustrate the effect of *laxity* on the scheduler: since there is only one thread *causing* page faults, and one thread *satisfying* them, no client ever has more than one transaction outstanding. In this case the EDF algorithm unmodified by laxity would allow each client exactly one transaction per period.

Notice further that the length of any laxity line never exceeds 10ms, the value specified above, and that the use of laxity does not impact the deadlines of clients.

7.2.2 Paging Out

The second experiment is designed to illustrate the overall performance and isolation achieved when multiple domains are paging out data to different parts of the same disk. The test application operates with a slightly modified stretch driver in order to achieve this effect — it "forgets" that pages have a copy on disk and hence never pages in during a page fault. The other parameters are as for the previous experiment. The resulting progress is plotted on the left hand side of Figure 8.

As can been seen, the domains once again proceed roughly in proportion, although overall throughput is much reduced. The reason for this is in the detailed USD scheduler trace on the right hand side of Figure 8. This clearly shows that almost every transaction is taking on the order of 10ms, with some clearly taking an additional rotational delay. This is most likely due to the fact that individual transactions are separated by a small amount of time, hence preventing the driver from performing any transaction coalescing.

One may also observe the fact that the client with the smallest slice (which is 25ms) tends to complete three transactions (totalling more than 25ms) in some periods, but then will obtain less time in the following period. This is since we employ a roll-over accounting scheme: clients are al-

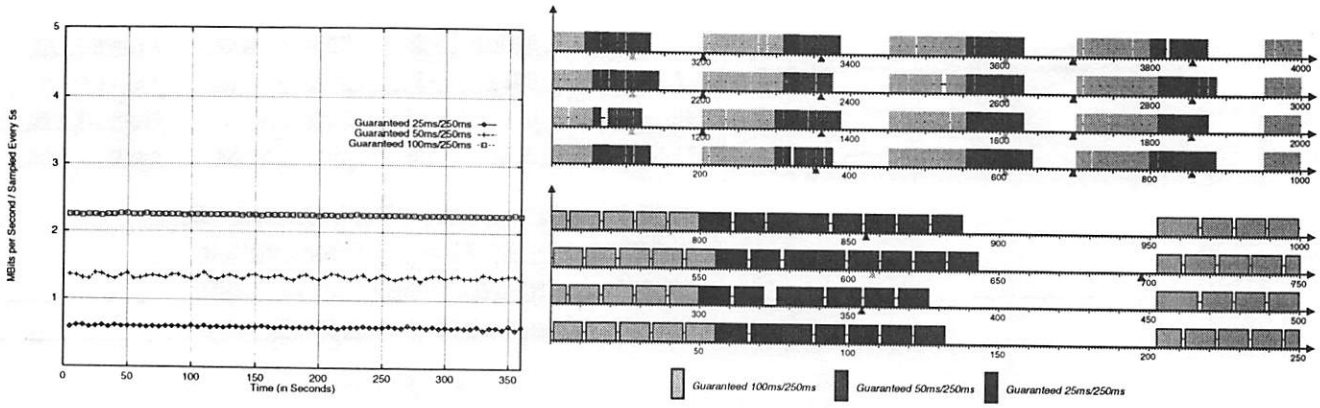


Figure 8: Paging Out (*lhs* shows sustained bandwidth, *rhs* shows a USD Scheduler Trace)

lowed to complete a transaction if they have a reasonable amount of time remaining in the current period. Should their transaction take more than this amount of time, the client will end with a negative amount of remaining time which will count against its next allocation.

Using this technique prevents an application deterministically exceeding its guarantee. It is not perfect — since it allows jitter to be introduced into the schedule — but it is not clear that there is a better way to proceed without intimate knowledge of the disk caching and scheduling policies.

7.3 File-System Isolation

The final experiment presented here adds another factor to the equation: a client domain reading data from another partition on the same disk. This client performs significant *pipelining* of its transaction requests (i.e. it trades off additional buffer space against disk latency), and so is expected to perform well. For homogeneity, its transactions are each the same size as a page.

The file-system client is guaranteed 50% of the disk (i.e. 125ms per 250ms). It is first run on its own (i.e. with no other disk activity occuring) and achieves the sustained bandwidth shown in the left hand side of Figure 9. Subsequently it was run again, this time concurrently with two paging applications having 10% and 20% guarantees respectively. The resulting sustained bandwidth is shown in the right hand side of Figure 9.

As can be seen, the throughput observed by the file-system client remains almost exactly the same despite the addition of two heavily paging applications.

8 Conclusion

This paper has presented the idea of *self-paging* as a technique to provide Quality of Service to applications. Experiments have shown that it is possible to accurately isolate the effects of application paging, which allows the coexistence of paging along with time-sensitive applications. Most of the VM system is provided by unprivileged user-level modules which are explicitly and dynamically linked, thus supporting extensibility.

Performance can definitely be improved. For example, the $3\mu s$ overhead in user-space trap-handling could probably be cut in half. Additionally the current stretch driver implementation is immature and could be extended to handle additional pipe-lining via a "stream-paging" scheme such as that described in [24].

A more difficult problem with the self-paging approach, however, is that of *global* performance. The strategy of allocating resources directly to applications certainly gives them more control, but means that optimisations for global benefit are not directly enforced. Ongoing work is looking at both centralised and devolved solutions to this issue.

Nonetheless, the result is promising: virtual memory techniques such as demand-paging and memory mapped files have proved useful in the commodity systems of the past. Failing to support them in the continuous media operating systems of the future would detract value, yet supporting them is widely perceived to add unacceptable unpredictability. Self-paging offers a solution to this dilemma.

Acknowledgments

I should like to express my extreme gratitude to Paul Barham who encouraged me to write this paper, and wrote the tools to log and post-process the USD scheduler traces. Without his help, this paper would not have been possible.

I would also like to thank the anonymous reviewers and my shepherd, Paul Leach, for their constructive comments and suggestions.

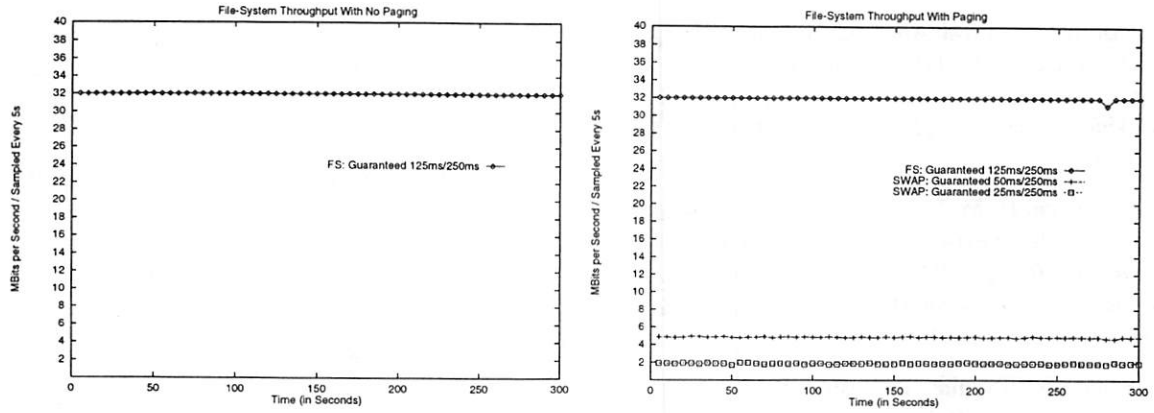


Figure 9: File-System Isolation

Availability

The Nemesis Operating System has been developed as part of the Pegasus II project, supported by the European Communities' ESPRIT programme. A public release of the source code will be made in 1999.

References

[1] E. Hyden. *Operating System Support for Quality of Service*. PhD thesis, University of Cambridge Computer Laboratory, February 1994.

[2] T. Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge Computer Laboratory, April 1995.

[3] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, September 1996. Article describes state in May 1995.

[4] M. B. Jones, P. J. Leach, R. P. Draves, and III J. S. Barrera. Modular Real-Time Resource Management in the Rialto Operating System. In *Processings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 12–17, May 1995.

[5] M. B. Jones, III J. S. Barrera, A. Forin, P. J. Leach, D. Rosu, and M. Rosu. An Overview of the Rialto Real-Time Architecture. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 249–256, September 1996.

[6] M. B. Jones, D. Rosu, and M. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, pages 198–211, October 1997.

[7] D. Mosberger. *Scout: A Path-Based Operating System*. PhD thesis, University of Arizona, Department of Computer Science, 1997.

[8] J. Nieh and M. S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, pages 184–197, October 1997.

[9] D. Engler, S. K. Gupta, and F. Kaashoek. AVM: Application-Level Virtual Memory. In *Processings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1995.

[10] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles*, December 1995.

[11] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, October 1996.

[12] R. P. Draves, G. Odinak, and S. M. Cutshall. The Rialto Virtual Memory System. Technical Report MSR-TR-97-04, Microsoft Research, Advanced Technology Division, February 1997.

[13] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[14] P. R. Barham. A Fresh Approach to Filesystem Quality of Service. In *7th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 119–128, St. Louis, Missouri, USA, May 1997.

[15] P. Shenoy and H. M. Vin. Cello: A Disk Scheduling Framework for Next-Generation Operating Systems. In *Proceedings of ACM SIGMETRICS'98, the International Conference on Measurement and Modeling of Computer Systems*, June 1998.

[16] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th ACM SIGOPS Symposium on Operating Systems Principles*, pages 63–76, November 1987.

[17] K. Harty and D. R. Cheriton. Application-Controlled Physical Memory using External Page-Cache Management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 187–197, October 1992.

[18] Y. A. Khalidi and M. N. Nelson. The Spring Virtual Memory System. Technical Report SMLI TR-93-9, Sun Microsystems Laboratories Inc., February 1993.

[19] J. Liedtke. On μ-Kernel Construction. In *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles*, pages 237–250, December 1995.

[20] Y. Endo, J. Gwertzman, M. Seltzer, C. Small, K. A. Small, and D. Tang. VINO: the 1994 Fall Harvest. Technical Report TR-34-94, Center for Research in Computing Technology, Harvard University, December 1994. Compilation of six short papers.

[21] D. R. Cheriton and K. J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 179–194, November 1994.

[22] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[23] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 96–107, April 1991.

[24] G. E. Mapp. *An Object-Oriented Approach to Virtual Memory Management*. PhD thesis, University of Cambridge Computer Laboratory, January 1992.

[25] P. Cao. *Application-Controlled File Caching and Prefetching*. PhD thesis, Princeton University, January 1996.

[26] C. A. Thekkath and H. M. Levy. Hardware and Software Support for Efficient Exception Handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 110–121, October 1994.

[27] D. Engler, F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles*, December 1995.

[28] R. Black, P. Barham, A. Donnelly, and N. Stratford. Protocol Implementation in a Vertically Structured Operating System. In *Proceedings of the 22nd Conference on Local Computer Networks*, pages 179–188, November 1997.

[29] M. F. Kaashoek, D. R. Engler, G. R. Granger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, pages 52–65, October 1997.

[30] B. N. Bershad, D. Lee, T. H. Romer, and J. Bradley Chen. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 158–170, October 1994.

[31] Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK. *ANSAware/RT 1.0 Manual*, March 1995.

[32] P. Barham. *Devices in a Multi-Service Operating System*. PhD thesis, University of Cambridge Computer Laboratory, July 1996.

[33] R. J. Black. *Explicit Network Scheduling*. PhD thesis, University of Cambridge Computer Laboratory, April 1995.

[34] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System

Ben Gamsa* Orran Krieger† Jonathan Appavoo* Michael Stumm*

*Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada
{ben,jonathan,stumm}@eecg.toronto.edu

†IBM T.J. Watson Research Center
Yorktown Heights, New York
okrieg@us.ibm.com

Abstract

We describe the design and implementation of Tornado, a new operating system designed from the ground up specifically for today's shared memory multiprocessors. The need for improved locality in the operating system is growing as multiprocessor hardware evolves, increasing the costs for cache misses and sharing, and adding complications due to NUMAness. Tornado is optimized so that locality and independence in application requests for operating system services—whether from multiple sequential applications or a single parallel application—are mapped onto locality and independence in the servicing of these requests in the kernel and system servers. By contrast, previous shared memory multiprocessor operating systems all evolved from designs constructed at a time when sharing costs were low, memory latency was low and uniform, and caches were small; for these systems, concurrency was the main performance concern and locality was not an important issue.

Tornado achieves this locality by starting with an object-oriented structure, where every virtual and physical resource is represented by an independent object. Locality, as well as concurrency, is further enhanced with the introduction of three key innovations: (*i*) *clustered objects* that support the partitioning of contended objects across processors, (*ii*) a *protected procedure call* facility that preserves the locality and concurrency of IPC's, and (*iii*) a new locking strategy that allows all locking to be encapsulated within the objects being protected and greatly simplifies the overall locking protocols. As a result of these techniques, Tornado has far better performance characteristics, particularly for multithreaded applications, than existing commercial operating systems. Tornado has been fully implemented and runs both on Toronto's NUMAchine hardware and on the SimOS simulator.

1 Introduction

Traditional multiprocessor operating systems, including those commercially available today (e.g., IBM's AIX, Sun's Solaris, SGI's IRIX, HP's HP/UX), all evolved from designs when multiprocessors were generally small, memory latency relative to processor speeds was comparatively low, memory sharing costs were low, memory access costs were uniform, and caches and cache lines were small.[1] The primary performance concern for these systems was to maximize concurrency, primarily by identifying contended locks and breaking them up into finer-grained locks.

Modern multiprocessors, such as Stanford's Dash and Flash systems, Toronto's NUMAchine, SGI's Origin, HP/Convex Exemplar, and Sun's larger multiprocessors, introduce new serious performance problems because of (*i*) higher memory latencies due to faster processors and more complex controllers, (*ii*) large write sharing costs, (*iii*) large secondary caches, (*iv*) large cache lines that give rise to false sharing, (*v*) NUMA effects, and (*vi*) larger system sizes, that stress the bottlenecks in the system software and uncover new ones. These characteristics all require that the system software be optimized for locality, something that was not necessary in the past. In addition to maximizing temporal and spatial locality as required for uniprocessors, optimizing for locality in the context of modern multiprocessors also means:

- minimizing read/write and write sharing so as to minimize cache coherence overheads,
- minimizing false sharing, and
- minimizing the distance between the accessing processor and the target memory module (in the case of NUMA multiprocessors).

To understand the importance of locality in modern multiprocessors, consider a simple counter (for example, a performance counter or a reference count) with multiple threads concurrently updating it. Figure 1 shows the performance of different implementations of such a counter (in cycles per update) when run on SimOS simulating a 16 processor, 4 processor-per-node NUMA multiprocessor. In these experiments, sufficient delays are introduced between counter updates to ensure that the counter is not contended. The shared counter scales well if a system is simulated with hardware parameters set to those typical of ten years ago (third curve from the bottom), but it

[1] In fact, most of these systems evolved from uniprocessor operating system designs.

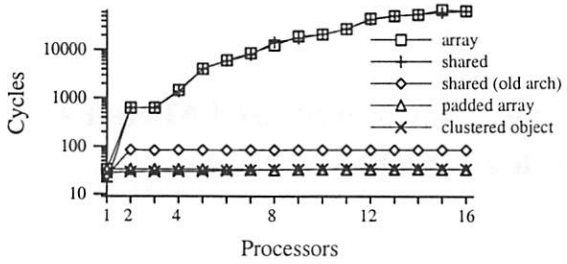


Figure 1: *Cycles per update of a counter (log scale).*

performs orders of magnitude worse when simulating a modern shared memory multiprocessor, such as NUMA-chine (top-most curve).[2] The counter performs no better when the counter is converted to an array with each processor updating its own counter individually (complicating the extraction of the total counter value) because of false sharing. Good performance can be achieved only by additionally padding each array entry to the secondary cache line size, or by applying the clustered object techniques described later in this paper (bottom two curves).

In general, data structures that may have been efficient in previous systems, and might even possess high levels of concurrency, are often inappropriate in modern systems with high cache miss and write sharing costs. Moreover, as the counter example above demonstrates, a single poorly constructed component accessed in a critical path can have a serious performance impact. While the importance of locality has been recognized by many implementors of shared memory multiprocessor operating systems, it can be extremely difficult to retrofit locality into existing operating system structures. The partitioning, distribution, and replication of data structures as well as the algorithmic changes needed to improve locality are difficult to isolate and implement in a modular fashion, given a traditional operating system structure.

The fact that existing operating system structures have performance problems, especially when supporting parallel applications, is exemplified in Figure 2, which shows the results of a few simple micro-benchmarks run on a number of commercial multiprocessor operating systems.[3] For each commercial operating system considered, there is a significant slowdown when simple operations are issued in parallel that should be serviceable completely independently of each other.

In this paper, we describe the design and implementation of Tornado, a new shared memory multiprocessor operating system that was designed from the ground up with the primary overriding design principle of mapping any locality and independence that might exist in OS requests from applications to locality and independence in the servicing of these requests in the operating system kernel and servers. More specifically, Tornado is designed to service all OS requests on the same processor they are issued on, and to handle requests to different resources without accessing any common data structures and without acquiring any common locks. As a result, Tornado does not exhibit the difficulties of the aforementioned systems (see Figure 2). Moreover, we found that we could achieve this by using a small number of relatively simple techniques in a systematic fashion. As a result, Tornado has a simpler structure than other multiprocessor operating systems, and hence can be more easily maintained and optimized.

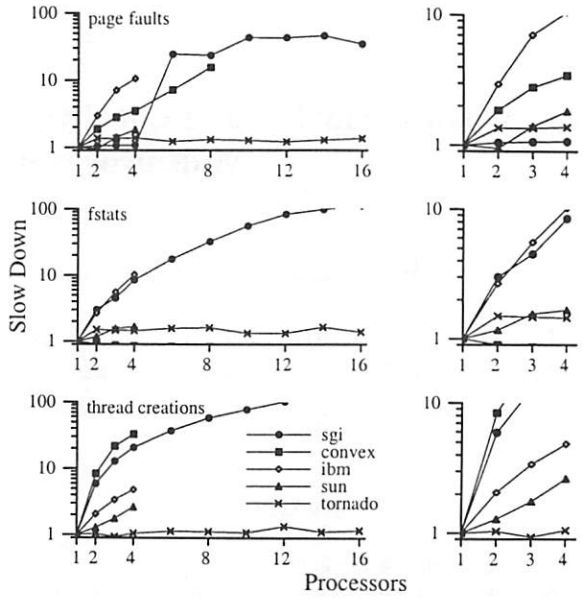


Figure 2: *Normalized cost (log scale) of simultaneously performing on n processors: n in-core page faults (top), n fstats (middle), and n thread creations/deletions (bottom) for 5 commercial shared memory multiprocessor operations systems and for Tornado. A full description of these experiments can be found in Section 7.*

Tornado uses an object-oriented approach, where every virtual and physical resource in the system is represented by an independent object, ensuring natural locality and independence for all resources. Aside from its object-oriented structure, Tornado has three additional innovations that help maximize locality (as well as concurrency). First, *Clustered Objects* allow an object to be partitioned into *representative objects*, where independent requests on different processors are handled by different representatives of the object in the common case. Thus, simultaneous requests from a parallel application to a single virtual resource (i.e., page faults to different pages of the same memory region) can be handled efficiently preserving as much locality as possible.

Second, the Tornado *Protected Procedure Call* facility maps the locality and concurrency in client requests to the servicing of these requests in the kernel and sys-

[2] Also, although not shown, the number of cycles required per update varies greatly for the different threads due to NUMA effects.

[3] While micro-benchmarks are not necessarily a good measure of overall performance, these results do show that the existing systems can have performance problems.

tem servers, and yet performs competitively with the best uniprocessor IPC facilities. Thus, repeated requests to the same server object (such as a read for a file) are serviced on the same processor as the client thread, and concurrent requests are automatically serviced by different server threads without any need for data sharing or synchronization to start the server threads.

Finally, Tornado uses a semi-automatic garbage collection scheme that facilitates localizing lock accesses and greatly simplifies locking protocols. As a matter of principle, all locks are internal to the objects (or more precisely their representatives) they are protecting, and no global locks are used. In conjunction with clustered object structures, the contention on a lock is thus bounded by the clients of the representative being protected by the lock. With the garbage collection scheme, no additional (existence) locks are needed to protect the locks internal to the objects. As a result, Tornado's *locking strategy* results in much lower locking overhead, simpler locking protocols, and can often eliminate the need to worry about lock hierarchies.

The foundation of the system architecture is Tornado's object-oriented design strategy, described in Section 2. This is followed by a description of the three key components discussed above: clustered objects (Section 3), locking (Section 5), and protected procedure calls (Section 6), with a short interlude to consider memory allocation issues in Section 4. Although we focus primarily on the Tornado kernel, it is important to note that these components are also used in the implementation of the Tornado system servers.

Tornado is fully implemented (in C++), and runs on our 16 processor NUMAchine [14, 31] and on the SimOS simulator [27]; it supports most of the facilities (e.g., shells, compilers, editors) and services (pipes, TCP/IP, NFS, file system) one expects. Experimental results that demonstrate the performance benefits of our design are presented in Section 7, followed by an examination of related work in Section 8 and concluding remarks in Section 9.

2 Object-oriented structure

Operating systems are driven by the requests of applications on virtual resources such as virtual memory regions, network connections, threads, address spaces, and files. To achieve good performance on a multiprocessor, requests to different virtual resources should be handled independently; that is, without accessing any shared data structures and without acquiring any shared locks. One natural way to accomplish this is to use an object-oriented strategy, where each resource is represented by a different object in the operating system.

As an example, Figure 3 shows, in a slightly simplified form, the key objects used for memory management in Tornado. On a page-fault, the exception is delivered to the *Process* object for the thread that faulted. The *Process* object maintains the list of mapped memory regions in the process's address space, which it searches to identify the responsible *Region* object to forward the request to. The region translates the fault address into a file offset, and forwards the request to the *File Cache Manager* (FCM) for the file backing that portion of the address space. The FCM checks if the file data is currently cached in memory. If it is, then the address of the corresponding physical page frame is returned to the *Region*, which makes a call to the *Hardware Address Translation* (HAT) object to map the page, and then returns. Otherwise, the FCM requests a new physical page frame from the DRAM manager, and asks the *Cached Object Representative* (COR) to fill the page from a file. The COR then makes an upcall to the corresponding file server to read in the file block. The thread is re-started when the file server returns with the required data.

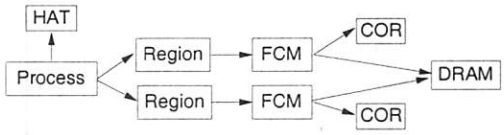


Figure 3: *The key memory management object relationships in Tornado.*

This example illustrates the advantage of employing an object-oriented approach. In the performance critical case of an in-core page fault, all objects invoked are specific to either the faulting process or the file(s) backing that process; the locks acquired and data structures accessed are internal to the objects. Hence, when different processes are backed by (logically) different files, there are no potential sources of contention. Also, for processes running on different processors, the operating system will not incur any communication misses when handling their faults. In contrast, many operating systems maintain a global page cache or a single HAT layer which can be a source of contention and offers no locality.

Localizing data structures in the Tornado fashion results in some new implementation and policy tradeoffs. For example, without a global page cache, it is difficult to implement global policies like a clock replacement algorithm in its purest form. Memory management in Tornado is based on a working set policy (similar to that employed by NT [10]), and most decisions can be made local to FCMs.

In Tornado, most operating system objects have multiple implementations, and the client or system can choose the best implementation to use at run time. The implementation can be specific to the degree of sharing, so implementing an object with locking protocols and data structures that scale is only necessary if the object is widely shared. As a result, a lower overhead implementation can be used when scalability is not required. We have found the object-oriented structure of Tornado to greatly simplify its implementation, allowing us to initially implement services using only simple objects with limited

concurrency, improving the implementation only when performance (or publication) required it. In the future we expect to be able to dynamically change the objects used for a resource.

Although our object-oriented structure is not the only way to get the benefits of locality and concurrency, it is a natural structuring technique and provides the foundation for other Tornado features such as the clustered object system [24] and the building block system [1].

3 Clustered Objects

Although the object-oriented structure of Tornado can help reduce contention and increase locality by mapping independent resources to independent objects, some components, such as a File Cache Manager (FCM) for a shared file, a Process object for a parallel program, or the system DRAM Manager, may be widely shared and hence require additional locality enhancing measures.

Common techniques used to reduce contention for heavily shared components include replication, distribution, and partitioning of objects. For example, for the (performance) counter discussed in the introduction, full distribution of the counter is used to ensure each processor can independently update its component, at the cost of making the computation of the true current value (i.e., the sum of all elements) more complicated. As another example, consider the thread dispatch queue. If a single list is used in a large multiprocessor system, it may well become a bottleneck and contribute significant additional cache coherency traffic. By partitioning the dispatch queue so that each processor has a private list, contention is eliminated.[4]

These types of optimizations have been applied before in other systems, but generally in an ad hoc manner, and only to a few individual components. The goal of Tornado's clustered object system is to facilitate the application of these techniques as an integral part of a system's overall design [24].

3.1 Overview

A clustered object presents the illusion of a single object, but is actually composed of multiple component objects, each of which handles calls from a specified subset of the processors (see Figure 4). Each component object represents the collective whole for some set of processors, and is thus termed a clustered object *representative*, or just *rep* for short. All clients access a clustered object using a common clustered object reference (that logically refers to the whole), with each call to the clustered object automatically directed to the appropriate local representative.

The internal structure of a clustered object can be defined in a variety of ways. One aspect of this variety

[4]However, it too has complications, due to the difficulty of ensuring good load balancing and respecting system-wide priorities.

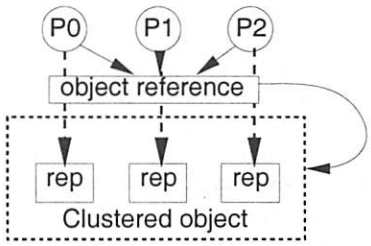


Figure 4: *An abstract depiction of a clustered object.*

is the *degree of clustering*. There might be one rep for the entire system, one rep per processor, or one rep for a cluster of neighboring processors. For example, in the case of the memory management subsystem of Tornado, the Cached Object Representative (COR) has a single rep that is shared across all processors (thus acting as a regular shared object) since it is read-mostly and only invoked when file I/O is required (which occurs relatively infrequently). The Region is also read-mostly, but is on the critical path for all page faults, and hence can benefit from partial replication, with one rep per cluster of processors. On the other hand, the FCM maintains the state of the pages of a file cached in memory, and hence can benefit from a partitioning strategy, where the hash table for the cache is distributed across a number of reps (at least for files that are being widely shared).

With multiple reps per object, it is necessary to keep them consistent. A variety of strategies are possible, including invalidation and update protocols. Coordination between reps of a given object can occur either through shared memory or a form of remote execution in Tornado called remote PPCs (described later in Section 6). Although shared memory is generally more efficient for fine-grained operations, it can sometimes be cheaper to incur the cost of the remote execution facility and perform the operation local to the data, if there is the possibility of high contention, or if there is a large amount of data to be accessed. With an efficient remote execution facility, the tradeoff point can be as low as a few tens of cache misses.

The use of clustered objects has several benefits. First, it facilitates the types of optimizations commonly applied on multiprocessors, such as replication or partitioning of data structures and locks. Second, it preserves the strong interfaces of object-oriented design so that clients need not concern themselves with the location or organization of the reps, and just use clustered object references like any other object reference. All complexity resides in the internal implementation of the clustered objects and the clustered object system.

Third, clustered objects enable incremental optimizations. Initially, a clustered object might be implemented with just one rep serving requests from all processors. This implementation would be almost identical to a corresponding non-clustered implementation. If performance requirements demand it, then the clustered object can be

successively optimized independently of the rest of the system.

Fourth, several different implementations of a clustered object may exist, each with the same external interface, but each optimized for different usage patterns.

Finally, the clustered object system supports dynamic strategies where the specific type of representative can be changed at run time based on the type and distribution of requests.

3.2 Application of Clustered Objects

In our implementation, large-grain objects, like FCM, Region, and Thread objects, are candidates for clustered objects, rather than smaller objects like linked lists. One example that illustrates many of the benefits of our approach is the Process object.

Because a Process can have multiple threads running on multiple processors and most of the Process object accesses are read-only, the Process clustered object is replicated to each processor the process has threads running on. On other processors, a simple rep is used that redirects all calls to one of the full reps. Some fields, like the base priority, are updated by sending all modifications to the home rep and broadcasting an update to all the other reps. Other components, like the list of memory Regions in the process, are updated on demand as each rep references the Region for the first time, reducing the cost of address space changes when Regions are not widely shared. If threads of the program are migrated, the corresponding reps of the Process object are migrated with them to maintain locality.

As an illustration of some of the tradeoffs with clustered objects, consider Figure 5(a) which shows the performance of page fault handling for a multithreaded program (more details on the experiments are provided in Section 7). With multiple reps for the Process object, the Region list is replicated and page faults can be processed with full concurrency, compared to the simple shared case where a lock on the Region list creates a bottleneck. The clustered object structure effectively splits the Process lock among the representatives, allowing each rep to lock its copy independently, thus eliminating both the write-sharing of the lock and its contention. Although it would appear that a similar effect could be achieved with a reader-writer lock protecting a single shared Process object, the lock is normally held for such a short duration that the overheads of the reader-writer lock (including the write-sharing it entails) would overshadow any concurrency benefits.

However, by replicating the Process object, other operations, such as deleting a memory Region, become more expensive (see Figure 5(b)) because of the need to keep the Process object reps consistent. This tradeoff is generally worthwhile for the Process object, since page faults are much more common than adding or deleting regions.

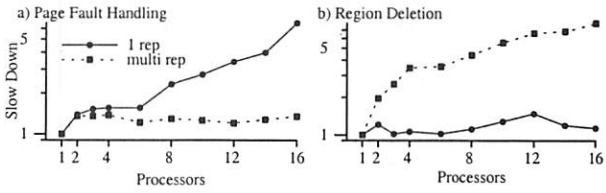


Figure 5: *Comparison of the performance of a Process clustered object with one rep vs. a Process clustered object with n reps (one per processor) for (a) in-core page fault handling and (b) Region destruction*

3.3 Clustered Object implementation

The key to the implementation of clustered objects is the use of per-processor translation tables. For each clustered object the tables maintain a pointer to the rep responsible for handling method invocations for the local processor. A clustered object reference is just a pointer into the table, with the extra level of indirection through the local table providing the mechanism for locating the local rep. (A clustered object call thus requires just one extra instruction.) By having each per-processor copy of the table located at the same virtual address, a single pointer into the table will refer to a different entry (the rep responsible for handling the invocation) on each processor.

Because it is generally unknown *a priori* which reps will be needed when and where, reps are typically created on demand when first accessed. This is accomplished by requiring each clustered object to define a miss handling object[5] and by initializing all entries in the translation tables to point to a special global miss handling object (see Figure 6). When an invocation is made for the first time on a given processor, the global miss handling object is called. Because it knows nothing about the intended target clustered object, it blindly saves all registers and calls the clustered object's miss handler to let it handle the miss. The object miss handler then, if necessary, creates a rep and installs a pointer to it in the translation table; otherwise, if a rep already exists to handle method invocations from that processor, a pointer to it is installed in the translation table. In either case, the object miss handler returns with a pointer to the rep to use for the call. The original call is then restarted by the global miss handler using the returned rep, and the call completes as normal, with the caller and callee unaware of the miss handling work that took place. This whole process requires approximately 150 instructions,[6] which although non-negligible, is still inexpensive enough to be used as a general purpose mechanism for triggering dynamic actions beyond just inserting a rep into the table.[7]

[5] Default miss handlers are provided for the common cases.

[6] All references to instructions in the paper are for MIPS instructions, which should be comparable to most other RISC processors. Results from full experimental tests are presented in Section 7.

[7] The entry can also be pre-filled to avoid the cost of the miss, should the rep structure be known in advance.

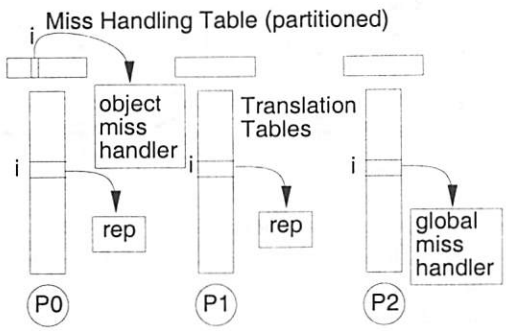


Figure 6: *Overview of clustered object implementation. Clustered object i has been accessed on P0 and P1, where reps have been installed; P2 has not yet accessed object i.*

Miss handling requires that all public methods of a clustered object be virtual and that a standard error code type be returned to allow the global miss handler to return an error when necessary. This restriction has not been a problem for us, since Tornado uses an object-oriented structure, and since most clustered objects have abstract base classes to allow different implementations.

To allow the global miss handler to locate the clustered object, the clustered object miss handler is installed at creation time in a global miss handling object table. This table is not replicated per-processor, but instead is partitioned, with each processor receiving a fraction of the total table. The global miss handler can then use the object reference that caused the miss to locate the clustered object's miss handler. This interaction with the clustered object system adds approximately 300 instructions to the overhead of object creation and destruction. Again, although significant, we feel the various benefits of clustered objects make this cost reasonable, especially if the ratio of object calls to object creation is high, as we expect it to be.

Finally, the translation tables are likely to be sparsely populated, because (*i*) there can be a large number of clustered objects (tens of thousands per processor), (*ii*) the translation table on each processor has to be large enough to handle all objects created anywhere in the system, and (*iii*) many clustered objects are only accessed on the processor on which they are created. As a result, the translation tables reside in virtual memory (even in the kernel), with pages only allocated when needed. However, instead of paging out pages when memory runs low, our implementation simply discards victim pages, since the table is really just a cache of entries, with the miss handlers of the clustered objects keeping track of the existence and location of the reps (i.e., they maintain the backing copy).[8]

[8] As an optimization, it would make sense to compress the victim pages to a fixed size compression table (i.e., second-level cache), since many of the pages are sparsely populated, but we have not yet implemented this.

4 Dynamic Memory Allocation

For the clustered object approach to be effective, it requires a facility that supports processor-local memory allocation. The memory allocator must be efficient, highly concurrent, and maximize locality both in its internal design and in the memory it returns on allocation.

Tornado's initial memory allocation facility used pools of memory per processor to support locality in memory allocations, using a design similar to that of [23]. However, we found that per-processor pools were not sufficient due to cache line false sharing problems that occur with small block allocations. We addressed this by providing a separate per-processor pool for small blocks that are intended to be accessed strictly locally.[9] Additionally, to address NUMA locality issues, the Tornado allocator partitions the pools of free memory into clusters. Although this requires an extra check on each free to determine the home cluster of the target block, this adds only three instructions to the critical path, most of which can be hidden in super-scalar processors. Finally, to support user-level allocations, instead of disabling interrupts, we use an optimized locking protocol that takes advantage of the common availability of load-linked/store-conditional instructions in today's processors to reduce locking overhead to just four instructions.

The allocator requires only 16 instructions for common case allocation, and 21 instructions for the common case deallocation (including checks for remote frees and overfull free lists), while providing locality, concurrency, and efficiency.[10]

5 Synchronization

There are two kinds of locking issues in most systems: those related to concurrency control with respect to modifications to data structures, which we refer to simply as *locking*, and those related to protecting the existence of the data structures; i.e., providing *existence guarantees* to ensure the data structure containing the variable is not deallocated during an update. We discuss each in turn.

5.1 Locking

One of the key problems with respect to locking is its overhead. In addition to the basic instruction overhead of locking, there is the cost of the extra cache coherence traffic due to the intrinsic write-sharing of the lock. With Tornado's object-oriented structure it is natural to encapsulate all locking within individual objects. This helps reduce the scope of the locks and hence limits contention.

[9] Another approach would been to make the minimum block size of the allocator the same as the cache line size, but at 128 bytes, this can increase internal fragmentation considerably.

[10] Although difficult to accurately measure on a super-scalar out-of-order processor, on a MIPS R10000 processor the cost of a malloc/free pair was about 7 cycles over and above that of a null function call.

Moreover, the use of clustered objects helps limit contention further by splitting single objects into multiple representatives thus limiting contention for any one lock to the number of processors sharing an individual representative. This allows us to optimize for the uncontended case. We use highly efficient spin-then-block locks, that require only two dedicated bits[11] from any word (such as the lower bits of an aligned pointer), at a total cost of 20 instructions for a lock/unlock pair in the uncontended case.[12]

5.2 Existence guarantees

Providing existence guarantees is likely the most difficult aspect of concurrency control. The traditional way of eliminating races between one thread trying to lock an object and another deallocating it, is to ensure that all references to an object are protected by their own lock, and all the references are used only while holding the lock on the reference. The disadvantage of this approach is that the reference lock in turn needs its own protector lock, with the pattern repeating itself until some root object that can never be deallocated. This results in a complex global lock hierarchy that must be strictly enforced to prevent deadlock, and it encourages holding locks for long periods of time while operations on referenced objects (and their referenced objects) are performed. For example, in the page fault example, the traditional approach would require holding a lock on the process object for the duration of the page fault, solely to preserve the continued existence of the Regions it references.[13]

For Tornado, we decided to take a somewhat different approach, using a semi-automatic garbage collection scheme for clustered objects. With this approach, a clustered object reference can be safely used at any time, whether any locks are held or not, even as the object is being deleted. This simplifies the locking protocol, often eliminating the need for a lock completely (for example, for read-only objects). It also removes the primary reason for holding locks across object invocations, increasing modularity and obviating the need for a lock hierarchy in most cases.

5.3 Garbage collection implementation

The key idea in the implementation of our semi-automatic garbage collection scheme is to distinguish between what we call temporary references and persistent references. Temporary references are all clustered object references that are held privately by a single thread, such as references on a thread's stack; these references are all implicitly destroyed when the thread terminates. In contrast, persistent references are those stored in (shared) memory; they can be accessed by multiple threads and might survive beyond the lifetime of a single thread.

The distinction between temporary and persistent references is used to divide clustered object destruction into three phases. In the first phase, the object ensures that all persistent references to it have been removed. This is part of the normal cleanup procedure required in any system; when an object is to be deleted, references to the object must be removed from lists and tables and other objects.

In the second phase, the clustered object system insures that all temporary references have been eliminated. To achieve this, we chose a scheme that is simple and efficient. It is based on the observation that the kernel and system servers are event driven and that the service times for those events are relatively short. In describing our scheme, first consider the uniprocessor case. The number of operations (i.e., calls to the server from some external client thread) currently active is maintained in a per-processor counter; the counter is incremented every time an operation is started and decremented when the operation completes. Thus, when the count is zero, we know there can be no live temporary references to any object on that processor, and phase two ends.[14] A concern with this approach is that there is no guarantee that the count of live threads will ever actually return to zero, which could lead to a form of starvation. However, since system server calls tend to be short, we do not believe this to be a problem in practice.[15]

For the multiprocessor case, we need to also consider threads running on other processors with temporary references to the object. Each clustered object can easily know which set of processors can access it, because the first access to an object on a processor always results in a translation table miss, and any reference stored in an object can be accessed only by the set of processors that have already made an access to the object. Hence, the set of processors can be determined when cleaning up the persistent references by determining which processors have objects with a persistent reference to the target clustered object and forming the union of the set. We use a circulating token scheme to determine that the per-processor counters have reached zero on each processor of the target processor set, with the token visiting each processor

[11] One bit indicates if the lock is held and the other indicates if there are queued threads. A separate shared hash table is used to record the list of waiting threads.

[12] We measured the lock time on an R10000 processor in a manner similar to memory allocation, and found it cost about 10 cycles over the cost of a pair of null function calls.

[13] A different approach altogether is to use lock-free concurrency control. However, practical algorithms often require additional instructions not currently found on modern processors, and they have their own difficulties in dealing with memory deallocation [15, 18]

[14] This is a much stronger requirement than actually required; it would, for example, suffice to ensure that all threads have completed that were active when object destruction was initiated, but that would require keeping track of much more information that just a raw count.

[15] One approach under consideration is to swap the active count variable periodically, so that the count of new calls is isolated from the count of previous calls. More careful investigation is still required.

that potentially holds references to the object being destroyed. When a processor receives the token it waits until its count of active threads goes to zero, before passing it on to the next processor. When the token comes back to the initiating processor it knows that the active count has gone to zero on all processors since it last had the token.[16]

Finally, when all temporary references have been eliminated, the clustered object can be destroyed (i.e., its reps can be destroyed, their memory released, and the clustered object entry freed).

Unfortunately, we have not yet tuned this code, so it currently requires approximate 270 instructions to fully deallocate an object once it has been handed to the cleanup system.

6 Interprocess communication

In a microkernel system like Tornado that relies on client-server communication, it is important to extend the locality and concurrency of the architecture beyond the internal design of individual components to encompass the full end-to-end design. For example, in the memory management subsystem, each page fault is an implicit call from the faulting thread to the process object; each call by the Cached Object Representative (COR) to its corresponding file object in the file system is another cross-process object call; and each call from the file system to the device driver object is another. Concurrency and locality in these communications are crucial to maintaining the high performance that exists within the various Tornado subsystems and servers.

Tornado uses a Protected Procedure Call (PPC) model [13], where a call from a client object to a server object acts like a clustered object call that crosses from the protection domain of the client to that of the server, and then back on completion. The key advantages of the PPC model are that: (*i*) client requests are always serviced on their local processor; (*ii*) clients and servers share the processor in a manner similar to handoff scheduling; and (*iii*) there are as many threads of control in the server as client requests. This also means that any client-specific state maintained in the server can be held local to the client, reducing unnecessary cache traffic. For example, for page faults to memory-mapped files that require I/O, all of the state concerning the file that the client has mapped can be maintained in data structures local to the thread that is accessing the mapped file. In some sense, this is like extending the Unix trap-to-kernel process model to all servers (and the kernel for Tornado), but without needing to dedicate resources to each client, as all clients use the same port to communicate to a given server, including the kernel. The PPC model is thus a key component to enabling locality and concurrency within servers.

[16] To deal with scalability, there can be multiple tokens circulating, covering different subsets of processors.

To support cross-process clustered object calls, a stub generator is provided that generates stubs based on the public interface of a clustered object. The clustered object system also includes support in the way of a few extra bits in the object translation table that identify those clustered objects that can accept external calls. To ensure references invoked from an external source are valid, the PPC subsystem checks to ensure they fall within the translation table, are properly aligned, and pass the security bits check. This makes it easy to use clustered object references, in conjunction with the identity of the target server, as a global object reference. As a result, cross-process clustered object calls are made directly to the local rep of the target object, providing the same locality and concurrency benefits for cross-process calls as for in-process calls.

The PPC facility also supports cross-*processor* communication among clustered objects. Remote PPCs are used primarily for device interactions, for function shipping between the reps of a clustered object to coordinate their state where preferred over shared memory access (i.e., data shipping), and for operating on processor-local exception-level state (such as the per-processor PPC thread cache).

6.1 PPC implementation

The implementation of our PPC facility involves on-demand creation of server threads to handle calls and the caching of the threads for future calls. To maximize performance in a multiprocessor environment, state information about a PPC endpoint (a Tornado port), including a cache of ready threads, is maintained on a per-processor basis. This allows a full call and return to be completed without the need for any locks (beyond disabling interrupts as normally happens as part of the PPC trap) or accesses to shared data (in the common case that the port cache is not empty).

For each server that has previously been accessed on a processor, the processor maintains a list of worker threads to handle calls,which grows and shrinks according to the demand on the server on the processor. A PPC call involves just a trap, a couple of queue manipulations to dequeue the worker and enqueue the caller on the worker, and a return-from-trap to the server, with a similar sequence for a return PPC call. Parameter passing registers are left untouched by the call sequence and hence are implicitly passed between client and server.

On first use, as well as for the uncommon case of there being no workers available on the port, the call is redirected to a special *MetaPort*, whose sole purpose is to handle these special cases. The workers for this port have resources pre-reserved for them so that these redirected calls will always succeed.

Although the design is primarily targeted at maximizing concurrency, a common case call and return requires only 372 instructions, including 50 instructions for user-level register save and restore, 14 for stubs, 49 for the

clustered object security checking code, and 30 instructions for debugging. Stacks can optionally be mapped dynamically among all threads, which adds 62 instructions to the base latency, but allows the memory to be reused across servers, minimizing the cache footprint. This leaves 167 instructions for the core cost of a PPC call and return, which compares favorably to the corresponding cost of 158 instructions for two one-way calls for one of the fastest uniprocessor IPC systems running on the same (MIPS R4000 based) processor [20]. In addition, up to 8KB of data can be exchanged between client and server (in both directions) through remapping a region for an additional cost of only 82 instructions.

The final component of the PPC system, remote PPCs, are like regular PPCs with a pair of remote interrupts on the call and return to connect the two sides. One key difference, however, is that a full context switch is required on both sides for both the call and return. One natural optimization we have not yet applied would be to have the caller spin in the trap handler for a few microseconds before calling the scheduler in case the remote PPC call completes quickly, avoiding the overhead of the two context switches on the calling side. Still more expensive than we would like, the remote PPC call/return pair requires approximately 2200 instructions (including 1300 for two remote interrupt exchanges), plus the cost of four cache transfers (a pair for each of the remote PPC call and return exchanges).[17]

7 Experimental results

The results presented in this paper are based on both hardware tests on a locally developed 16 processor NUMAchine prototype [14, 31] and the SimOS simulator from Stanford [27]. The NUMAchine architecture consists of a set of *stations* (essentially small, bus-based multiprocessors) connected by a hierarchy of rings. It uses a novel selective broadcast mechanism to efficiently handle invalidations, as well as broadcast data. The test machine has 4 stations, each with 4 processors and a memory module (for a total of 16 processors), connected by a single ring. The final machine will have 48 processors with a two-level hierarchy of rings. The processors are 150MHz MIPS R4400 processors with 16K I/D L1 cache and 1MB L2 cache, all direct mapped. The bus and rings are 64 bits wide and clocked at 40MHz,[18] giving a bandwidth of 320MB/s for each link. The key latencies for the system are: 15 cycles for the secondary cache, 270 for local memory, and approximately 370 cycles for remote memory.

Although the simulator does not model precisely the same architecture as our hardware (in particular the interconnect and the coherence protocol implementation

[17] We expect to reduce this to about 600 instructions when optimizations already applied to the local case are applied to the remote call path.

[18] The final version will run at 50MHz.

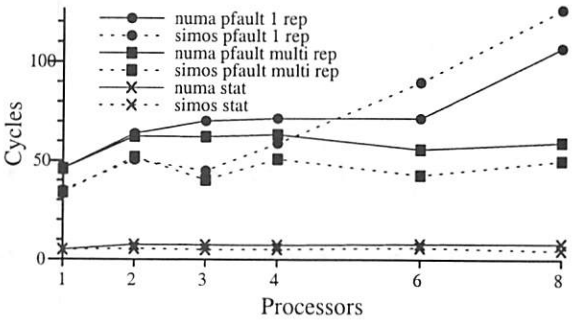


Figure 7: *Comparison of SimOS vs. NUMAchine for various benchmarks.*

are different), it does model a similar NUMA architecture based on the MIPS R4400 processor with component sizes, latencies, and bandwidths configured similar to those of NUMAchine. In addition, the simulator has been validated by previous researchers and by ourselves using both micro- and macro-benchmarks. For example, Figure 7 shows the results of a number of different tests run on SimOS and our hardware. (The tests are discussed in more detail below.) Although the exact cycle counts are not identical, the general trends match closely, allowing us to reasonably evaluate the effect of various tradeoffs in the Tornado architecture.

In this section we examine the performance of the individual components presented in this paper, and then look at some higher level microbenchmarks run under both Tornado and other current multiprocessor systems.

7.1 Component results

Figure 8(a) shows the results of a number of concurrent stress tests on the components described in this paper; namely, dynamic memory allocation (n threads malloc and free), clustered object miss handling (n threads invoke independent clustered objects not in the table), clustered object garbage collection (n threads trigger garbage collection), and protected procedure calling (n threads call a common clustered object in another address space). Results are collected over a large number of iterations and averaged separately for each thread. Figure 8(a) includes the average time in cycles across all threads, as well as range bars indicating the range of thread times.

These results show that memory allocation and miss handling perform quite well although there is a lot of variance across the threads for the garbage collection and PPC tests. As the number of processors increases, the range remains consistent, but the overall average slowly increases. If we compare the results from NUMAchine to those on SimOS, shown in Figure 8(b), we see the same sort of trend (except that it is worse under SimOS). However, running the same tests with SimOS set to simulate the caches with 4-way associativity, the results become almost perfectly uniform and flat (see Figure 8(c)). This indicates that the cause of the variability is local

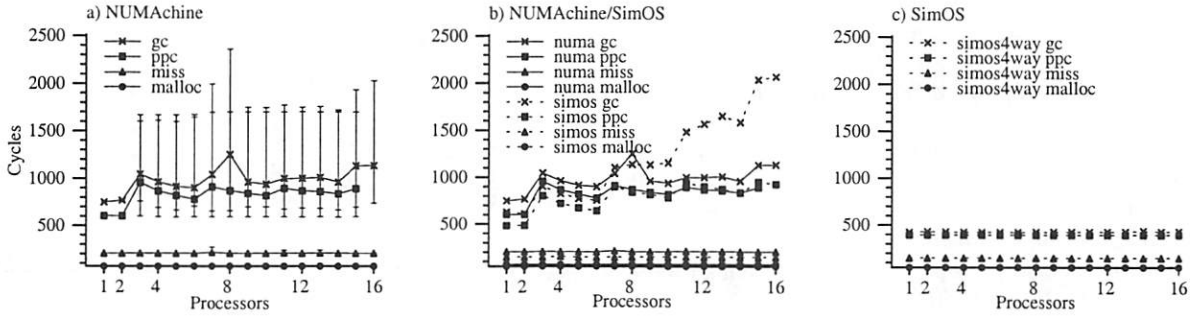


Figure 8: *Nanobenchmarks: Garbage collection (gc), Protected Procedure Calls (ppc), in-core page miss handling (miss), and dynamic memory allocation (malloc), run under Tornado on NUMAchine (numa) and under Tornado on SimOS (simos). The left figure (a) shows the average number of cycles required on NUMAchine for n threads with range bars indicating the range over all threads. The middle figure (b) shows the average cycles required on NUMAchine and SimOS. The right figure (c) shows the average cycles required on SimOS configured with 4-way set associative caches.*

cache conflicts—caused by multiple data structures occasionally mapping to the same cache block on some processors—and is not due to some unforeseen sharing.

7.2 Microbenchmarks

To evaluate the effectiveness of the Tornado design at a level above the individual components, we ran a few multiprocessor operating system stress tests. The microbenchmarks are composed of three separate tests: thread creation, in-core page faults, and file stat, each with n worker threads performing the operation being tested:

Thread Creation Each worker successively creates and then joins with a child thread (the child does nothing but exit).

In-Core Page Fault Each worker thread accesses a set of in-core unmapped pages in independent (separate mmap) memory regions.

File Stat Each worker thread repeatedly fstats an independent file.

Each test was run in two different ways; multithreaded and multiprogrammed. In the multithreaded case, the test was run as described above. In the multiprogrammed tests, n instances of the test were started with one worker thread per instance. In all cases, the tests were run multiple times in succession and the results were collected after a steady state was reached. Although there was still a high variability from run to run and between the different threads within a run, the overall trend was consistent.

Figure 9(a) shows normalized results for the different tests on NUMAchine. Because all results are normalized against the uniprocessor tests, an ideal result would be a set of perfectly flat lines at 1. Overall, the results demonstrate good performance, since the slowdown is usually less than 50 percent. However, as with the component tests, there is high variability in the results, which accounts for the apparent randomness in the graphs.

Similar results are obtained under SimOS. Figure 9(b) shows the raw times in microseconds for the multithreaded tests run under NUMAchine and SimOS. As with the component tests, setting SimOS to simulate 4-way associative caches smooths out the results considerably (Figure 9(c)).[19]

System	OS	# Cpus	Legend
UofT NUMAchine	Tornado	16	numa
SimOS NUMAchine	Tornado	16	simos
SimOS 4way[a]	Tornado	16	simos4way
SUN 450 UltraSparc II	Solaris 2.5.1	4	sun
IBM G30 PowerPC 604	AIX 4.2.0.0	4	ibm
SGI Origin 2000	IRIX 6.4	40[b]	sgi
Convex SPP-1600	SPP-UX 4.2	32[c]	convex

[a] simulated 4-way set associative cache
[b] Maximum used in experiments is 16
[c] Maximum used in experiments is 8

Table 1: *Platforms on which micro-benchmarks where run.*

Operation	Thread Creation	Page Fault	File Stat
NUMAchine	15	46	5
Sun	178	19	3
IBM	691	43	3
SGI	11	21	2
Convex	84	56	5

Table 2: *Base costs, in microseconds, for thread creation, page fault handling, and file stating, with a single processor.*

To see how existing systems perform, we ran the same tests on a number of systems available to us (see Table 1, Table 2, and Figure 10). The results demonstrate a number of things. First, many of the systems do quite well on the multiprogrammed tests, reflecting the effort that has gone into improving multi-user throughput over the last 10–15 years. However, the results are somewhat mixed for the multithreaded tests. In particular, although SGI does extremely well on the multiprogrammed

[19] There is still an anomaly involving a single thread taking longer than the others that we have yet to track down. This pulls up the average at two processors, but has less effect on the average when there are more threads running.

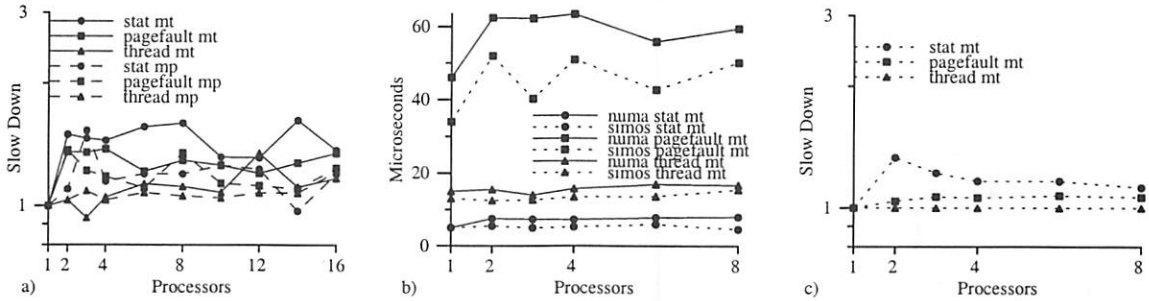


Figure 9: *Microbenchmarks: Cost of thread creation/destruction (thread), in-core page fault handling (pagefault), and file stat (stat) with n worker threads running either in one process (mt) or in n processes with one thread per process (mp). The left figure (a) shows slowdown relative to the uniprocessor case. The middle figure (b) shows the raw times in microseconds for the multithreaded tests on NUMAchine and SimOS, and the right figure (c) shows the slowdown of the multithreaded tests run on SimOS configured with 4-way set associative caches.*

tests, it does quite poorly on the multithreaded tests. This is particularly interesting when compared to results on an older bus-based, 6-processor SGI Challenge running IRIX 6.2 (not shown), where the multiprogrammed results are slightly worse and the multithreaded results are quite a bit better. Overall, Sun performs quite well with good multithreaded results and respectable multiprogrammed results; however, we only had a four processor system, so it is hard to extrapolate the results to larger systems.

One possible reason for poor performance is load balancing at a very fine granularity. For example, we suspect that the poor performance of AIX in the multiprogrammed thread creation experiment is due to a shared dispatch queue resulting in frequent thread migration. While load balancing is important, for most workloads systems like IRIX deal with it at a large granularity that does not require a shared dispatch queue.

7.3 Summary of results

While Tornado is still very much an active research project, the performance results obtained so far demonstrate the strengths of our basic design. The cycle counts provided throughout this paper for the costs of various operations demonstrates base performance competitive with commercial systems, which is important since scalability is of limited value if the base overhead is too large. In Section 7.1 we saw that the infrastructure of our system is highly scalable, including the IPC, clustered object, and the memory allocation facilities, providing the foundation for scalable system services. In Section 7.2 we saw that, for simple microbenchmarks, our system exhibits much better scalability than commercial systems. While microbenchmark results are generally considered a poor metric for comparison, the nearly perfect scalability of Tornado compares favorably to the large (e.g., 100X on 16 processors) slowdown for the commercial systems on the multithreaded experiments. It seems unlikely that this kind of a slowdown in the performance of such fundamental operations as page faults, thread creation, and file state does not have a large impact on application performance.

The disparity between the performance of the multithreaded and multiprogrammed results for the commercial systems suggests that locality and locking overhead on the operating system structures that represent a process is a major source of slowdown. The process is only a single example of a shared object, and we expect that experimentation will demonstrate that commercial systems exhibit slowdown (for even multiprogrammed experiments) when resources such as files or memory regions are shared. In our future work we will investigate the performance of Tornado when other resources are shared, and study the performance of our system for real applications.

8 Related work

A number of papers have been published on performance issues in shared-memory multiprocessor operating systems, mostly in the context of resolving specific problems in a specific system [5, 6, 9, 22, 26, 28]. These systems were mostly uniprocessor or small-scale multiprocessor systems trying to scale up to larger systems. Other work on locality issues in operating system structure were mostly either done in the context of earlier non-cache-coherent NUMA systems [8], or, as in the case of Plan 9, were not published [25]. Two projects that were aimed explicitly at large-scale multiprocessors were Hive [7], and the precursor to Tornado, Hurricane [30]. Both independently chose a clustered approach by connecting multiple small-scale systems to form either, in the case of Hive, a more fault tolerant system, or, in the case of Hurricane, a more scalable system. However, both groups ran into complexity problems with this approach and both have moved on to other approaches: Disco [4] and Tornado, respectively.

Clustered objects. Concepts similar to clustered objects have appeared in a number of distributed systems,

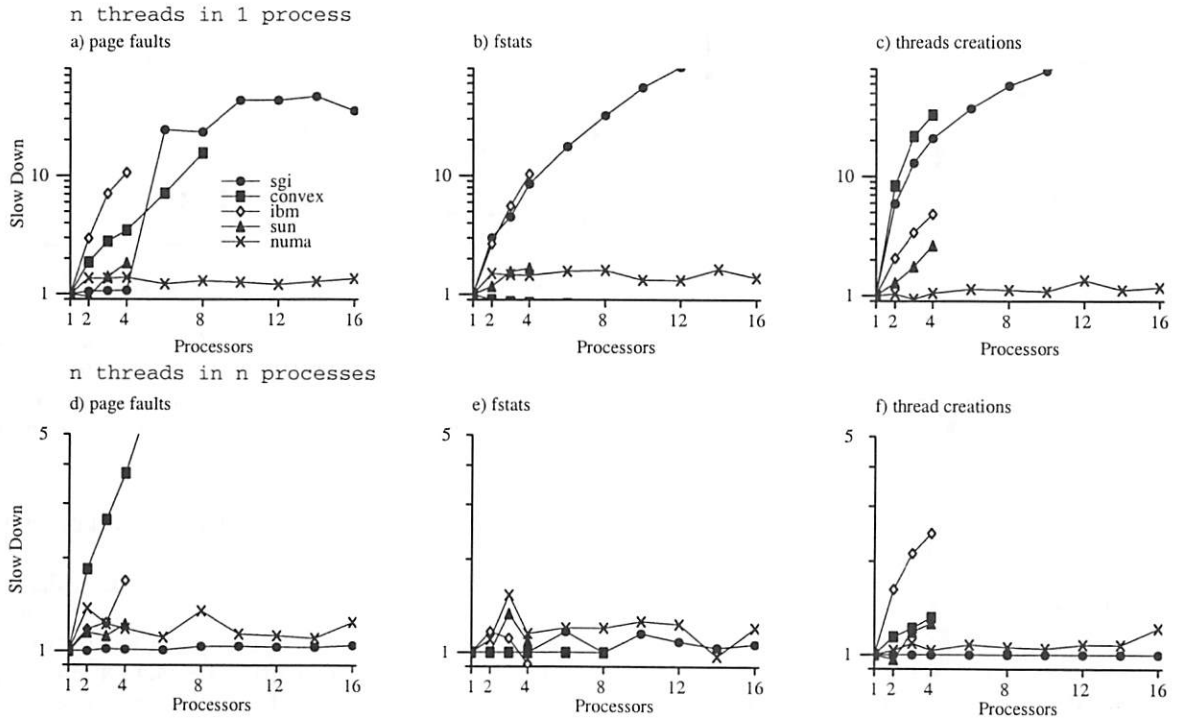


Figure 10: *Microbenchmarks across all tests and systems. The top row (a–c) depicts the multithreaded tests with n threads in one process. The bottom row (d–f) depicts the multiprogrammed tests with n processes, each with one thread. The leftmost set (a,d) depicts the slowdown for in-core page fault handling, the middle set (b,e) depicts the slowdown for file stat, and the rightmost set depicts the slowdown for thread creation/destruction. The systems on which the tests were run are: SGI Origin 2000 running IRIX 6.4, Convex SPP-1600 running SPP-UX 4.2, IBM 7012-G30 PowerPC 604 running AIX 4.2.0.0, Sun 450 UltraSparc II running Solaris 2.5.1, and NUMAchine running Tornado.*

most notably in Globe [19] and SOS [21]. In all these cases (including Tornado's clustered objects) the goal is to hide the distributed nature of the objects from the users of the objects while improving performance over a more naive centralized approach. However, the issues faced in a tightly coupled shared-memory multiprocessor are very different from those of a distributed environment. For example, communication is cheaper, efficiency (time and space) is of greater concern, direct sharing is possible, and the failure modes are simpler. Hence, the Tornado clustered object system is geared more strongly towards maximizing performance and reducing complexity than the other systems.

Dynamic memory allocation. Our dynamic memory allocation design borrows heavily from McKenney and Slingwine's design [23], which is one of the few published works on multiprocessor memory allocation, in particular for kernel environments. A survey paper by Wilson et al [33] covers many of the other schemes, but does not address multiprocessor or caching issues. Grunwald et al examined cache performance of allocation schemes [16] and suggest a number of techniques they felt would be most effective in dealing with locality issues. Most of these techniques can be found in the McKenney and Slingwine memory allocator (with a few additions in our own adaptation).

Synchronization. The topic of locking and concurrency control in general has a long history, as does garbage collection [32]. The relationship between locking and garbage collection is evident in some of the issues surrounding memory management for lock-free approaches [18]. Our garbage collection scheme is in some sense a hack, but works reasonably well in our environment. Although it is somewhat similar to IBM's patent 4809168, their scheme appears to target uniprocessors only and is less general than ours. The benefits for our locking protocol are particularly evident in large, complex software systems, where there are many developers with varying skill and experience in dealing with concurrency control.

Protected procedure call. The majority of research on performance conscious inter-process communication (IPC) is for uniprocessor systems. Excellent results have been reported for these systems, to the point where it has been argued that the IPC overhead has become largely irrelevant [2].[20] Although many results have been reported

[20] We do not agree with these arguments.

over the years on a number of different platforms, the core cost for a call-return pair (with similar functionality) is usually between 100 and 200 instructions [11, 12, 17, 20]. However, the Tornado PPC facility goes beyond the standard techniques used to optimize IPC performance, by optimizing for the multiprocessor case by eliminating locks and shared data accesses, and by providing concurrency to the servers.

The key previous work done in multiprocessor IPC was by Bershad et al [3], where excellent results were obtained on the hardware of the time. However, it is interesting that the recent changes in technology lead to design tradeoffs far different from what they used to be. The Firefly multiprocessor [29] on which Bershad's IPC work was developed has a smaller ratio of processor to memory speed, has caches that are no faster than main memory (used to reduce bus traffic), and uses an updating cache consistency protocol. For these reasons, Bershad found that he could improve performance by idling server processes on idle processors (if they were available), and having the calling process migrate to that processor to execute the remote procedure. This approach would be prohibitive in today's systems with high cost cache misses and invalidations.

9 Concluding Remarks

Tornado was built on our experience with the Hurricane operating system [30]. Hurricane employed a course grained approach to scalability, where a single large scale SMMP was partitioned into clusters of a fixed number of processors. Each cluster ran a separate instance of a small scale SMMP operating system, cooperatively providing a single system image. This approach is now being used in one form or another by several commercial systems, for example in SGI's Cellular IRIX. However, despite many of the positive benefits of this approach, we found that: (*i*) the traditional within-cluster structures exhibit poor locality which severely impacts performance on modern multiprocessors, (*ii*) the rigid clustering results in increased complexity as well as high overhead or poor scalability for some applications, and (*iii*) the traditional structures as well as the clustering strategy make it difficult to support the specialized policy requirements of parallel applications.

Tornado does not have these problems. The object-oriented nature of Tornado and its clustered objects allow any available locality and independence to be exploited, allow the degree of clustering to be defined on a per-object basis, and make it easier to explore policy and implementation alternatives. Moreover, the fine-grained, in-object locking strategy of Tornado has much lower complexity, lower overhead, and better concurrency.

As the adage goes, "any problem in computer science can be solved by an extra level of indirection."[21] In Tornado, the clustered object translation table provides this level of indirection, which we have found useful for several purposes. For example, it includes clustered object access control information for implementing IPC security, helps track accesses to objects in support of the garbage collection system, and supplants the need for many other global tables by allowing clients to directly use references to server objects, rather than using an identifier that must be translated to the target object on each call.

The Tornado object-oriented strategy does not come without cost, however. Overheads include: (*i*) virtual function invocation, (*ii*) the indirection through the translation table, and (*iii*) the intrinsic cost of modularity, where optimizations possible by having one component of the system know about the details of another are not allowed. Our experiences to-date suggest that these costs are low compared to the performance advantages of locality, and will over time grow less significant with the increasing discrepancy between processor speed and memory latency. However, more experimentation is required.

Our primary goal in developing Tornado was to design a system that would achieve high performance on shared-memory multiprocessors. We believe that the performance numbers presented in this paper illustrate that we have been successful in achieving this goal. A result that is just as important that we did not originally target was ease of development. The object-oriented strategy coupled with clustered objects makes it easier to first get a simple correct implementation of a new service and then incrementally optimize its performance later. Also, the locking protocol has made it much easier for inexperienced programmers to develop code, both because fewer locks have to be acquired and because objects will not disappear even if locks on them are released.

Tornado currently runs on SimOS and on a 16 processor prototype of NUMAchine. It supports most of the facilities (e.g., shells, compilers, editors) and services (pipes, TCP/IP, NFS, file system) one expects. We are starting to explore scalability issues, work on policies for parallel applications, and study how to clusterize objects in a semi-automated fashion. A sister project, the Kitchawan operating system at IBM T.J.Watson Research Center, employs many of the ideas from Tornado, and is additionally exploring fault containment, availability, portability and some of the other issues required for an industrial strength operating system.

Acknowledgments

Rob Ho implemented most of the microbenchmarks. Ron Unrau and Tarek Abdelrahmen helped run the performance experiments. Many helped with the implementation of Tornado, in particular: Derek DeVries, Daniel Wilk, and Eric Parsons. Comments by Marc Auslander, Paul Lu, Karen Reid, Ron Unrau and David Wheeler.

[21] In a private communication, Roger Needham attributes this to

rau, and our shepherd Rob Pike helped improve the paper. Finally, this work was funded in part by IBM Corp. and the Natural Sciences and Engineering Research Council (NSERC).

References

[1] M. Auslander, H. Franke, O. Krieger, B. Gamsa, and M. Stumm. Customization-lite. In *6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 43–48, 1997.

[2] B. Bershad. The increasing irrelevance of IPC performance for microkernel-based operating systems. In *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 205–212, 1992.

[3] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Computer Systems*, 8(1):37–55, February 1990.

[4] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. on Computer Systems*, 15(4):412–447, November 1997.

[5] M. Campbell et al. The parallelization of UNIX system V release 4.0. In *Proc. USENIX Technical Conference*, pages 307–324, 1991.

[6] J. Chapin, S. A. Herrod, M. Rosenblum, and A. Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In *Proc. ACM SIGMETRICS Intl. Conf. on Measurement and Modelling of Computer Systems*, 1995.

[7] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosio, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP)*, pages 12–25, 1995.

[8] E. M. Jr. Chaves, P. C. Das, T. J. Leblanc, B. D. Marsh, and M. L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–191, May 1993.

[9] D. R. Cheriton and K. J. Duda. A caching model of operating system kernel functionality. In *Proc. Symp. on Operating Systems Design and Implementation (OSDI)*, pages 179–193, 1994.

[10] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.

[11] D. R. Engler, M. F. Kaashoek, and Jr. O'Toole J. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th ACM Symp. on Operating Systems Principles*, pages 251–266, 1995.

[12] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proc. USENIX Technical Conference*, pages 97–114, 1994.

[13] B. Gamsa, O. Krieger, and M. Stumm. Optimizing IPC performance for shared-memory multiprocessors. In *Proc. ICPP*, pages 208–211, 1994.

[14] A. Grbic et al. Design and implementation of the NUMAchine multiprocessor. In *Proceedings of the 35rd DAC*, pages 66–69, 1998.

[15] M. Greenwald and D.R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Symp. on Operating System Design and Implementation*, pages 123–136, 1996.

[16] D. Grunwald, B. G. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, pages 177–186, 1993.

[17] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proc. USENIX Summer Technical Conference*, 1993.

[18] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. on Programming Languages and Systems*, 15(5):745–770, November 1993.

[19] P. Homburg, L. van Doorn, M. van Steen, A. S. Tanenbaum, and Wi. de Jonge. An object model for flexible distributed systems. In *Proc. of the 1st Annual ASCI Conference*, pages 69–78, 1995.

[20] T. Jaeger et al. Achieved IPC performance. In *6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.

[21] M. Makpangou, Y. Gourhant, J.P. Le Narzul, and M. Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, 1994.

[22] D. McCrocklin. Scaling Solaris for enterprise computing. In *Spring 1995 Cray Users Group Meeting*, 1995.

[23] P. E. McKenney and J. Slingwine. Efficient kernel memory allocation on shared-memory multiprocessor. In *Proc. USENIX Technical Conference*, pages 295–305, 1993.

[24] E. Parsons, B. Gamsa, O. Krieger, and M. Stumm. (De-)clustering objects for multiprocessor system software. In *Proc. Fourth Intl. Workshop on Object Orientation in Operating Systems (IWOOS95)*, pages 72–84, 1995.

[25] R. Pike. Personal communication.

[26] D. L. Presotto. Multiprocessor streams for Plan 9. In *Proc. Summer UKUUG Conf.*, pages 11–19, 1990.

[27] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. on Modeling and Computer Simulation*, 7(1):78–103, Jan. 1997.

[28] J. Talbot. Turning the AIX operating system into an MP-capable OS. In *Proc. USENIX Technical Conference*, 1995.

[29] C. P. Thacker and L. C. Stewart. Firefly: a multiprocessor workstation. In *Proc. 2nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 164–172, 1987.

[30] R. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9(1/2):105–134, 1995.

[31] Z. Vranesic et al. The NUMAchine multiprocessor. Technical Report CSRI-324, Computer Systems Research Institute, University of Toronto, 1995.

[32] P. R. Wilson. Uniprocessor garbage collection techniques. In *Intl. Workshop on Memory Management*. Springer-Verlag, 1992.

[33] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Intl. Workshop on Memory Management*. Springer-Verlag, 1995.

Interface and Execution Models in the Fluke Kernel

Bryan Ford Mike Hibler Jay Lepreau Roland McGrath Patrick Tullmann

Department of Computer Science, University of Utah

Abstract

We have defined and implemented a kernel API that makes every exported operation fully interruptible and restartable, thereby appearing atomic to the user. To achieve interruptibility, all possible kernel states in which a thread may become blocked for a "long" time are represented as kernel system calls, without requiring the kernel to retain any unexposable internal state.

Since all kernel operations appear atomic, services such as transparent checkpointing and process migration that need access to the complete and consistent state of a process can be implemented by ordinary user-mode processes. Atomic operations also enable applications to provide reliability in a more straightforward manner.

This API also allows us to explore novel kernel implementation techniques and to evaluate existing techniques. The Fluke kernel's single source implements either the "process" or the "interrupt" execution model on both uniprocessors and multiprocessors, depending on a configuration option affecting a small amount of code.

We report preliminary measurements comparing fully, partially and non-preemptible configurations of both process and interrupt model implementations. We find that the interrupt model has a modest performance advantage in some benchmarks, maximum preemption latency varies nearly three orders of magnitude, average preemption latency varies by a factor of six, and memory use favors the interrupt model as expected, but not by a large amount. We find that the overhead for restarting the most costly kernel operation ranges from 2–8% of the cost of the operation.

1 Introduction

An essential issue of operating system design and implementation is when and how one thread can block and relinquish control to another, and how the state of a thread suspended by blocking or preemption is represented in the system. This crucially affects both the kernel interface that represents these states to user code, and the fundamental internal organization of the kernel implementation. A central aspect of this internal structure is the execution model in which the kernel handles in-kernel events such as processor traps, hardware interrupts, and system calls. In the *process model*, which is used by traditional monolithic kernels such as BSD, Linux, and Windows NT, each thread of control in the system has its own kernel stack—the complete state of a thread is implicitly encoded its stack. In the *interrupt model*, used by systems such as V [8], QNX [19], and the exokernel implementations [13, 22], the kernel uses only one kernel stack per *processor*—thus for typical uniprocessor configurations, only one kernel stack. A thread in a process-model kernel retains its kernel stack state when it sleeps, whereas in an interrupt-model kernel threads must explicitly save any important kernel state before sleeping, since there is no stack implicitly encoding the state. This saved kernel state is often known as a *continuation* [11], since it allows the thread to "continue" where it left off.

In this paper we draw attention to the distinction between an interrupt-model *kernel implementation*—a kernel that uses only one kernel stack per processor, explicitly saving important kernel state before sleeping—and an "atomic" *kernel API*—a system call API designed to eliminate implicit kernel state. These two kernel properties are related but fall on orthogonal dimensions, as illustrated in Figure 1. In a purely atomic API, *all* possible states in which a thread may sleep for a noticeable amount of time are cleanly visible as a kernel entrypoint. For example, the state of a thread involved in any system call is always well-defined, complete, and immediately available for examination or modification by other threads; this is true even if the system call is long-running and consists of many stages. In general, this requires all system calls and exception handling mechanisms to be cleanly *interruptible* and *restartable*, in the same way that the instruction sets of modern processor architectures are cleanly interruptible and restartable. For purposes of readability, in the rest of this paper we will refer to an API with these properties as "atomic." We use this term because, from the user's perspective, no thread is ever in the middle of any system call.

We have developed a kernel, Fluke [16], which exports a purely atomic API. This API allows the complete state of any user-mode thread to be examined and modified by other user-mode threads without being arbi-

This research was largely supported by the Defense Advanced Research Projects Agency, monitored by the Dept. of the Army under contract DABT63–94–C–0058, and the Air Force Research Laboratory, Rome Research Site, USAF, under agreement F30602–96–2–0269.

Contact information: lepreau@cs.utah.edu, Dept. of Computer Science, 50 Central Campus Drive, Rm. 3190, University of Utah, SLC, UT 84112-9205. http://www.cs.utah.edu/projects/flux/.

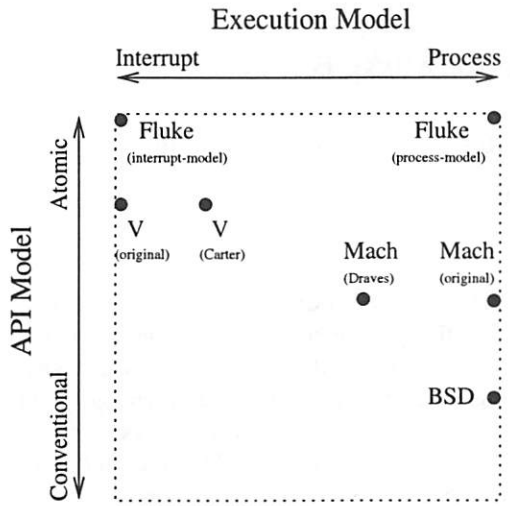


Figure 1: The kernel execution and API model continuums. V was originally a pure interrupt-model kernel but was later modified to be partly process-model; Mach was a pure process-model kernel later modified to be partly interrupt-model. Fluke supports either execution model via compile time options.

trarily delayed. Supporting a purely atomic API slightly increases the width and complexity of the kernel interface but provides important benefits to user-level applications in terms of the power, flexibility, and predictability of system calls.

In addition, Fluke supports *both* the interrupt and process execution models through a build-time configuration option affecting only a small fraction of the source, enabling a comparison between them. Fluke demonstrates that the two models are not necessarily as different as they have been considered to be in the past; however, they each have strengths and weaknesses. Some processor architectures inherently favor the process model and process model kernels are easier to make fully preemptible. Although full preemptibility comes at a cost, this cost is associated with preemptibility, not with the process model itself. Process model kernels use more per-thread kernel memory, but this is unlikely to be a problem in practice except for power-constrained systems. We show that while an atomic API is beneficial, the kernel's internal execution model is less important: an interrupt-based organization has a slight size advantage, whereas a process-based organization has somewhat more flexibility.

Finally, contrary to conventional wisdom, our kernel demonstrates that it is practical to use legacy process-model code even within interrupt-model kernels. The key is to run the legacy code in *user mode* but in the *kernel's address space*.

Our key contributions in this work are: (i) To present a kernel supporting a pure atomic API and demonstrate the advantages and drawbacks of this approach. (ii) To explore the relationship between an "atomic API" and the kernel's execution model. (iii) To present the first comparison between the two kernel *implementation* models using a kernel that supports both in pure form, revealing that the models are not necessarily as different as commonly believed. (iv) To show that it is practical to use process-model legacy code in an interrupt-model kernel, and to present several techniques for doing so.

The rest of the paper is organized as follows. In Section 2 we look at other systems, both in terms of the "atomicity" their API and in terms their execution models. In Section 3 we define the two models more precisely, and examine the implementation issues in each, looking at the strengths and weaknesses each model brings to a kernel. Fluke's atomic API is detailed in Section 4. In the following section, we present six issues of importance to the execution model of a kernel, with measurements based on different configurations of the same kernel. The final section summarizes our analysis.

2 Related Work

Related work is grouped into kernels with atomic or near-atomic system call APIs and work related to kernel execution models.

2.1 Atomic System Call API

The clean interruptibility and restartability of *instructions* is now recognized as a vital property of all modern processor architectures. However, this has not always been the case; as Hennessy and Patterson state:

> This last requirement is so difficult that computers are awarded the title *restartable* if they pass that test. That supercomputers and many early microprocessors do not earn that badge of honor illustrates both the difficulty of interrupts and the potential cost in hardware complexity and execution speed. [18]

Since kernel system calls appear to user-mode code essentially as an extension of the processor architecture, the OS clearly faces a similar challenge. However, few operating systems have met this challenge nearly as thoroughly as processor architectures have.

For example, the Unix API [20] distinguishes between "short" and "long" operations. "Short" operations such as disk reads are made non-interruptible on the assumption that they will complete quickly enough that the delay will not be noticeable to the application, whereas "long" operations are interruptible but, if interrupted, must be restarted manually by the application. This distinction is arbitrary and has historically been the source of numerous practical problems. The case of disk reads from an NFS server that has gone down is a well-known instance of this problem: the arbitrarily long delays caused by the network makes it inappropriate to treat the read operation

as "short," but on the other hand these operations cannot simply be changed to "long" and made interruptible because existing applications are not written with the expectation of having to restart file reads.

The Mach API [1] implements I/O operations using IPC; each operation is divided into an RPC-style request and reply stage, and the API is designed so that the operation can be cleanly interrupted after the request has been sent but before the reply has been received. This design reduces but does not eliminate the number of situations in which threads can get stuck in states that are not cleanly interruptible and restartable. For example, a common remaining case is when a page fault occurs while the kernel is copying the IPC message into or out of the user's address space; the IPC operation cannot be cleanly interrupted and restarted at this point, but handling the page fault may involve arbitrary delays due to communication with other user-mode servers or even across a network. KeyKOS [6] comes very close to solving this problem by limiting all IPC operations to transfer at most one page of data and performing this data transfer atomically; however, in certain corner-case situations it gains promptness by sacrificing correctness.[1] Amoeba [26] allows one user-mode process (or *cluster* in Amoeba terminology) to "freeze" another process for debugging purposes, but processes cannot be frozen in certain situations such as while waiting for an acknowledgement from another network node. V [8, 32] allows one process to examine and modify the state of another, but the retrieved state is incomplete, and state modification is only allowed if the target process is awaiting an IPC reply from the modifying process.

In scheduler activations [2], user threads have no kernel state at all when they are neither running on a processor in user-mode, nor blocked in the kernel on a system call or page fault. However, threads blocked in system calls have complex state that is represented by a "scheduler activation" kernel object just as it would be by a kernel thread object; that state is not available to the user.

The Cache Kernel [9] and the Aegis [13] and Xok [22] exokernels implement atomic kernel interfaces by restricting the kernel API to managing extremely low-level abstractions so that none of the kernel system calls ever have any reason to block, and therefore they avoid the need for handling restarts or interruptions. Although this is a perfectly reasonable and compelling design strategy, the somewhat higher-level Fluke API demonstrates that strict atomicity can be implemented even in the presence of more complex operations which are not inherently idempotent.

On the other side of the spectrum, the Incompatible Time Sharing (ITS) operating system [12], developed in the 1960s and 1970s at MIT for the DEC PDP-6 and PDP-10 computers, demonstrated the feasibility of implementing a fully atomic API at a much higher levels of abstraction than Fluke, implementing facilities such as process control, file systems, and networking. ITS allowed all system calls to be cleanly interrupted and restarted, representing all aspects of a suspended computation in the contents of a thread's user-mode registers: in fact, this property was a central principle of the system's design and substantial effort was made in the implementation to achieve it. An unpublished memo [4] describes the design and implementation in detail, though to our knowledge no formally published work has previously identified the benefits of an atomic API and explored the implementation issues.

There are several systems which use concepts similar to Fluke's atomic system call API in different areas of operating systems. Quicksilver [17] and Nonstop [3] are transactional operating systems; in both of these systems, the kernel provides primitives for maintaining transactional semantics in a distributed system. In this way, transactional semantics are provided for high-level services such as file operations even though the basic kernel operations on which they are built may not be. The VINO [29] kernel uses transactions to maintain system integrity when executing untrusted software extensions downloaded into the kernel. These transactions make graft invocations appear atomic, even though invocations of the basic kernel API are not wrapped in transactions.

2.2 Kernel Execution Models

Many existing kernels have been built using either the interrupt or the process model internally: for example, most Unix systems use the process model exclusively, whereas QNX [19], the Cache Kenrel, and the exokernels use the interrupt model exclusively. Other systems such as Firefly's Taos [25, 28] were designed with a hybrid model where threads often give up their kernel stacks in particular situations but can retain them as needed to simplify the kernel's implementation. Minix [30] used kernel threads to run process-model kernel activities such as device driver code, even though the kernel "core" used the interrupt model. The V kernel was originally organized around a pure interrupt model, but was later adapted by Carter [7] to allow multiple kernel stacks while handling page faults. The Mach 3.0 kernel [1] was taken in the opposite direction: it was originally created in the process model, but Draves [10, 11] later adapted it to use a partial interrupt model by adding continuations in key locations in the kernel and by introducing a "stack handoff" mechanism. However, not all kernel stacks for suspended

[1] If the client's data buffer into which an IPC reply is to be received is paged out by a user-mode memory manager at the time the reply is made, the kernel simply discards the reply message rather than allowing the operation to be delayed arbitrarily long by a potentially uncooperative user-mode pager. This usually was not a problem in practice because most paging in the system is handled by the kernel, which is trusted to service paging requests promptly.

threads were eliminated. Draves et. al. also identified the optimization of *continuation recognition*, which exploits explicit continuations to recognize the computation a suspended thread will perform when resumed, and do part or all of that work by mutating the thread's state without transferring control to the suspended thread's context. Though it was used to good effect within the kernel, user-mode code could not take advantage of this optimization technique because the continuation information was not available to the user. In Fluke, the continuation is explicit in the user-mode thread state, giving the user a full, well-defined picture of the thread's state.

The ITS system used the process model of execution, each thread always having a private kernel stack that the kernel switched to and from for normal blocking and preemption. To ensure the API's atomicity guarantee, implementations of system calls were required to explicitly update the user register state to reflect partial completion of the operation, or to register special cleanup handlers to do so for a system call interrupted during a block. Once a system call blocked, an interruption would discard all context except the user registers,[2] and run these special cleanup handlers. The implementation burden of these requirements was eased by the policy that each user memory page touched by system call code was locked in core until the system call completed or was cleaned up and discarded.

We are not aware of any previous kernel that simultaneously supported both the "pure" interrupt model and the "pure" process model through a compile-time configuration option.

3 The Interrupt and Process Models

An essential feature of operating systems is managing many computations on a smaller number of processors, typically just one. Each computation is represented in the OS by a thread of control. When a thread is suspended either because it blocks awaiting some event or is preempted when the scheduler policy chooses another thread to run, the system must record the suspended thread's state so that it can continue operation later. The way an OS kernel represents the state of suspended threads is a fundamental aspect of its internal structure.

In the *process model* each thread of control in the system has its own kernel stack. When a thread makes a system call or is interrupted, the processor switches to the thread's assigned kernel stack and executes an appropriate handler in the kernel's address space. This handler may at times cause the thread to go to sleep waiting for some event, such as the completion of an I/O request; at these times the kernel may switch to a different thread having its own separate kernel stack state, and then switch back later when the first thread's wait condition is satisfied. The important point is that each thread retains its kernel stack state even while it is sleeping, and therefore has an implicit "execution context" describing what operation it is currently performing. Threads may even hold kernel resources, such as locks or allocated memory regions, as part of this implicit state they retain while sleeping.

An *interrupt model* kernel, on the other hand, uses only one kernel stack per processor—for typical uniprocessor kernels, just one kernel stack. This stack only holds state related to the *currently running* thread; no state is stored for sleeping threads other than the state explicitly encoded in its thread control block or equivalent kernel data structure. Context switching from one thread to another involves "unwinding" the kernel stack to the beginning and starting over with an empty stack to service the new thread. In practice, putting a thread to sleep often involves explicitly saving state relating to the thread's operation, such as information about the progress it has made in an I/O operation, in a *continuation* structure. This continuation information allows the thread to continue where it left off once it is again awakened. By saving the required portions of the thread's state, it essentially performs the function of the per-thread kernel stack in the process model but without the overhead of a full kernel stack.

3.1 Kernel Structure vs. Kernel API

The internal thread handling model employed by the kernel is not the only factor in choosing a kernel design. There tends to be a strong correlation between the kernel's execution model and the *kinds* of operations presented by the kernel to application code in the kernel's API. Interrupt-model kernels tend to export short, simple, atomic operations that don't require large, complicated continuations to be saved to keep track of a long running operation's kernel state. Process-model kernels tend to export longer operations with more stages because they are easy to implement given a separate per-thread stack and they allow the kernel to get more work done in one system call. There are exceptions, however; in particular, ITS used one small (40 word) stack per thread despite its provision of an atomic API. [5]

Thus, in addition to the execution model of the kernel itself, a distinction can be drawn between an atomic API, in which kernel operations are designed to be short and simple so that the state associated with long-running activities can be maintained mostly by the application process itself, and a conventional API, in which operations tend to be longer and more complex and their state is maintained by the kernel invisibly to the application. This stylistic difference between kernel API designs is

[2] Each ITS thread (or "job") also had a small set of "user variables" that acted as extra "pseudo-registers" containing additional parameters for certain system calls. Fluke uses a similar mechanism on the x86 because it has so few registers.

analogous to the "CISC to RISC" shift in processor architecture design, in which complex, powerful operations are broken into a series of simpler instructions with more state exposed through a wider register file.

Fluke exports a fully interruptible and restartable ("atomic") API, in which there are no implicit thread states relevant to, but not visible and exportable to application code. Furthermore, Fluke's implementation can be configured at compile-time to use either execution model in its pure form (i.e., either exactly one stack per processor or exactly one stack per thread); to our knowledge it is the first kernel to do so. In fact, it is Fluke's atomic API that makes it relatively simple for the kernel to run using either organization: the difference in the kernel code for the two models amounts to only about two hundred assembly language instructions in the system call entry and exit code, and about fifty lines of C in the context switching, exception frame layout, and thread startup code. This code deals almost exclusively with stack handling. The configuration option to select between the two models has no impact on the functionality of the API. Note that the current implementation of Fluke is not highly optimized, and more extensive optimization would naturally tend to interfere with this configurability since many obvious optimizations depend on one execution model or the other: e.g., the process model implementation could avoid rolling back and restarting in certain cases, whereas the interrupt model implementation could avoid returning through layers of function calls by simply truncating the stack on context switches. However, similar optimizations generally apply in either case even though they may manifest differently depending on the model, so we believe that despite this caveat, Fluke still provides a valuable testbed for analyzing these models. The API and implementation model properties of the Fluke kernel and their relationships are discussed in detail in the following sections.

4 Properties of an Atomic API

An atomic API provides four important and desirable properties: *prompt* and *correct* exportability of thread state, and full *interruptibility* and *restartability* of system calls and other kernel operations. To illustrate these basic properties, we will contrast the Fluke API with the more conventional APIs of the Mach and Unix kernels.

4.1 Promptness and Correctness

The Fluke system call API supports the extraction, examination, and modification of the state of any thread by any other thread (assuming the requisite access checks are satisfied). The Fluke API requires the kernel to ensure that one thread always be able to manipulate the state of another thread in this way without being held up indefinitely as a result of the target thread's activities or its interactions with other threads in the system. Such state manipulation operations can be delayed in some cases, but only by activities internal to the kernel that do not depend on the promptness of other untrusted application threads; this is the API's *promptness* requirement. For example, if a thread is performing an RPC to a server and is waiting for the server's reply, its state must still be promptly accessible to other threads without delaying the operation until the reply is received.

In addition, the Fluke API requires that, if the state of an application thread is extracted at an arbitrary time by another application thread, and then the target thread is destroyed, re-created from scratch, and reinitialized with the previously extracted state, the new thread must behave indistinguishably from the original, as if it had never been touched in the first place. This is the API's *correctness* requirement.

Fulfilling only one of the promptness and correctness requirements is fairly easy for a kernel to do, but strictly satisfying both is more difficult. For example, if promptness is not a requirement, and the target thread is blocked in a system call, then thread manipulation operations on that target can simply be delayed until the system call is completed. This is the approach generally taken by debugging interfaces such as Unix's `ptrace` and `/proc` facilities [20], for which promptness is not a primary concern—e.g., if users are unable to stop or debug a process because it is involved in a non-interruptible NFS read, they will either just wait for the read to complete or do something to cause it to complete sooner—such as rebooting the NFS server.

Similarly, if correctness is not an absolute requirement, then if one thread tries to extract the state of another thread at an inconvenient time, the kernel can simply return the target thread's "last known" state in hopes that it will be "good enough." This is the approach taken by the Mach 3.0 API, which provides a `thread_abort` system call to forcibly break a thread out of a system call in order to make its state accessible; this operation is guaranteed to be prompt, but in some cases may affect the state of the target thread so that it will not behave properly if it is ever resumed. To support process migration, the OSF later added a `thread_abort_safely` system call [27] which provides correctness, but at the expense of promptness.

Prompt and correct state exportability are required to varying degrees in different situations. For process control or debugging, correctness is critical since the target thread's state must not be damaged or lost; promptness is not as vital since the debugger and target process are under the user's direct control, but a lack of promptness is often perceptible and causes confusion and annoyance to the user. For conservative garbage collectors which must check an application thread's stack and registers for

pointers, correctness is not critical as long as the "last-known" register state of the target thread is available. Promptness, on the other hand, is important because without it the garbage collector could be blocked for an arbitrary length of time, causing resource shortages for other threads or even deadlock. User-level checkpointing, process migration, and similar services clearly require correctness, since without it the state of re-created threads may be invalid; promptness is also highly desirable and possibly critical if the risk of being unable to checkpoint or migrate an application for arbitrarily long periods of time is unacceptable. Mission critical systems often employ an "auditor" daemon which periodically wakes up and tests each of the system's critical threads and data structures for integrity, which is clearly only possible if a correct snapshot of the system's state is available without unbounded delay. Common user-mode deadlock detection and recovery mechanisms similarly depend on examination of the state of other threads. In short, although real operating systems often get away with providing unpredictable or not-quite-correct thread control semantics, in general promptness and correctness are highly desirable properties.

4.2 Atomicity and Interruptibility

One natural implication of the Fluke API's promptness and correctness requirements for thread control is that all system calls a thread may make must either appear completely *atomic*, or must be cleanly divisible into user-visible atomic stages.

An atomic system call is one that always completes "instantaneously" as far as user code is concerned. If a thread's state is extracted by another thread while the target thread is engaged in an atomic system call, the kernel will either allow the system call to complete, or will transparently abort the system call and roll the target thread back to its original state just before the system call was started. (This contrasts with the Unix and Mach APIs, for example, where user code is responsible for restarting interrupted system calls. In Mach, the restart code is part of the Mach library that normally wraps kernel calls; but there are intermediate states in which system calls cannot be interrupted and restarted, as discussed below.)

Because of the promptness requirement, the Fluke kernel can only allow a system call to complete if the target thread is not waiting for any event produced by some other user-level activity; the system call must be currently running (i.e., on another processor) or it must be waiting on some kernel-internal condition that is guaranteed to be satisfied "soon" without any user-mode involvement. For example, a short, simple operation such as Fluke's equivalent of `getpid` will always be allowed to run to completion; whereas sleeping operations such as `mutex_lock` are interrupted and rolled back.

While many Fluke system calls can easily be made atomic in this way, others fundamentally require the presence of intermediate states. For example, there is an IPC system call that a thread can use to send a request message and then wait for a reply. Another thread may attempt to access the thread's state after the request has been sent but before the reply is received; if this happens, the request clearly cannot be "un-sent" because it has probably already been seen by the server; however, the kernel can't wait for the reply either since the server may take arbitrarily long to reply. Mach addressed this scenario by allowing an IPC operation to be interrupted between the send (request) and receive (reply) operations, later restarting the receive operation from user mode.

A more subtle problem is page faults that may occur while transferring IPC messages. Since Fluke IPC does not arbitrarily limit the size of IPC messages, faulting IPC operations cannot simply be rolled back to the beginning. Additionally, the kernel cannot hold off all accesses to the faulting thread's state, since page faults may be handled by user-mode servers. In Mach, a page fault during an IPC transfer can cause the system call to block until the fault is satisfied (an arbitrarily long period).

Fluke's atomic API allows the kernel to update system call parameters in place in the user-mode registers to reflect the data transferred prior to the fault. Thus, while waiting for the fault to be satisfied both threads are left in the well-defined state of having transferred some data and about to start an IPC to transfer more. The API for Fluke system calls is directly analogous to the interface of machine instructions that operate on large ranges of memory, such as the block-move and string instructions on machines such as the Intel x86 [21]. The buffer addresses and sizes used by these instructions are stored in registers, and the instructions advance the values in these registers as they work. When the processor takes an interrupt or page fault during a string instruction, the parameter registers in the interrupted processor state have been updated to indicate the memory about to be operated on, and the program counter remains at the faulting string instruction. When the fault is resolved, simply jumping to that program counter with that register state resumes the string operation in the exact spot it left off.

Table 1 breaks the Fluke system call API into four categories, based on the potential length of each system call. "Trivial" system calls are those that will always run to completion without putting the thread to sleep. For example, Fluke's `thread_self` (analogous to Unix's `getpid`) will always fetch the current thread's identifier without blocking. "Short" system calls usually run to completion immediately, but may encounter page faults or other exceptions during processing. If an exception happens then the system call will roll back and restart. "Long" system calls are those that can be expected to sleep for an extended period of time (e.g., waiting on a

Type	Examples	Count	Percent
Trivial	thread_self	8	7%
Short	mutex_trylock	68	64%
Long	mutex_lock	8	7%
Multi-stage	cond_wait, IPC	23	22%
Total		107	100%

Table 1: Breakdown of the number and types of system calls in the Fluke API. "Trivial" system calls always run to completion. "Short" system calls usually run to completion immediately, but may roll back. "Long" system calls can be expected to sleep indefinitely. "Multi-stage" system calls can be interrupted at intermediate points in the operation.

condition variable). "Multi-stage" system calls are those that can cause the calling thread to sleep indefinitely and can be interrupted at various intermediate points in the operation.

Except for cond_wait and region_search—a system call which can be passed an arbitrarily large region of memory—all of the multi-stage calls in the Fluke API are IPC-related. Most of these calls simply represent different options and combinations of the basic send and receive primitives. Although all of these entrypoints could easily be rolled into one, as is done in Mach, the Fluke API's design gives preference to exporting several simple, narrow entrypoints with few parameters rather than one large, complex entrypoint with many parameters. This approach enables the kernel's critical paths to be streamlined by eliminating the need to test for various options. However, the issue of whether system call options are represented as additional parameters or as separate entrypoints is orthogonal to the issue of atomicity and interruptibility; the only difference is that if a multi-stage IPC operation in Fluke is interrupted, the kernel may occasionally modify the user-mode instruction pointer to refer to a different system call entrypoint in addition to updating the other user-mode registers to indicate the amount of data remaining to be transferred.

In [31] we more fully discuss the consequences of providing an atomic API. In summary, the purely atomic API greatly facilitates the job of user-level checkpointers, process migrators, and distributed memory systems. The correct, prompt access to all relevant kernel state of any thread in a system makes user-level managers themselves correct and prompt. Additionally, the clean, uniform management of thread state in an atomic API frees the managers from having to detect and handle obscure corner cases. Finally, such an API simplifies the kernel itself and is fundamental to allowing the kernel implementation easily to use either the process or the interrupt model; this factor will be discussed in Section 5.

Object	Description
Mutex	A kernel-supported mutex which is safe for sharing between processes.
Cond	A kernel-supported condition variable.
Mapping	Encapsulates an imported region of memory; associated with a Space (destination) and Region (source).
Region	Encapsulates an exportable region of memory; associated with a Space.
Port	Server-side endpoint of an IPC.
Portset	A set of Ports on which a server thread waits.
Space	Associates memory and threads.
Thread	A thread of control, associated with a Space.
Reference	A cross-process handle on a Mapping, Region, Port, Thread or Space. Most often used as a handle on a Port that is used for initiating client-side IPC.

Table 2: The primitive object types exported by the Fluke kernel.

4.3 Examples from Fluke

The Fluke kernel directly supports nine primitive object types, listed in Table 2. All types support a common set of operations including create, destroy, "rename," "point-a-reference-at," "get_objstate," and "set_objstate." Obviously, each type also supports operations specific to its nature; for example, a Mutex supports lock and unlock operations (the complete API is documented in [15]). The majority of kernel operations are transparently restartable. For example, port_reference, a "short" system call, takes a Port object and a Reference object and "points" the reference at the port. If either object is not currently mapped into memory[3], a page fault IPC will be generated by the kernel after which the reference extraction will be restarted. In all such cases page faults are generated very early in the system call, so little work is thrown away and redone.

Simple operations that restart after an error are fairly uninteresting. The more interesting ones are those that update their parameters, or even the system call entry point, to record partial success of the operation. The simplest example of this is the cond_wait operation which atomically blocks on a condition variable, releasing the associated mutex, and, when the calling thread is woken, reacquires the mutex. In Fluke, this operation is broken into two stages: the cond_wait portion and the mutex_lock. To represent this, before a thread sleeps on the condition variable, the thread's user-mode instruction pointer is adjusted to point at the mutex_lock entry point, and the mutex argument is put into the appropriate

[3] In Fluke, kernel objects are mapped into the address space of an application with the virtual address serving as the "handle" and memory protections providing access control.

register for the new entrypoint. Thus, if the thread is interrupted or awoken it will automatically retry the mutex lock and not the whole condition variable wait.

An example of a system call that updates its parameters is the `ipc_client_send` system call, which sends data on an already established IPC connection to a waiting server thread. The call might either be the result of an explicit invocation by user code, or its invocation could have been caused implicitly by the kernel due to the earlier interruption of a longer operation such as `ipc_client_connect_send`, which establishes a new IPC connection and then sends data across it. Regardless of how `ipc_client_send` was called, at the time of entry one well-defined processor register contains the number of words to transfer and another register contains the address of the data buffer. As the data are transferred, the pointer register is incremented and the word count register decremented to reflect the new start of the data to transfer and the new amount of data to send. For example, if an IPC tries to send 8,192 bytes starting from address `0x08001800` and successfully transfers the first 6,144 bytes and then causes a page fault, the registers will be updated to reflect a 2,048 byte transfer starting at address `0x08003000`. Thus, the system call can be cleanly restarted without redoing any transfers. The IPC connection state itself is stored as part of the current thread's control block in the kernel so it is not passed as an explicit parameter, though that state too is cleanly exportable through a different mechanism. Interfaces of this type are relatively common in Fluke, and the majority of the IPC interfaces exploit both parameter and program counter manipulation.

4.4 Disadvantages of an Atomic API

This discussion reveals several potential disadvantages of an atomic API:

Design effort required: The API must be carefully designed so that all intermediate kernel states in which a thread may have to wait indefinitely can be represented in the explicit user-accessible thread state. Although the Fluke API demonstrates that this can be done, in our experience it does take considerable effort. As a simple example, consider the requirement that updatable system call parameters be passed in registers. If instead parameters were passed on the user-mode stack, modifying one might cause a page fault—an indefinite wait—potentially exposing an inconsistent intermediate state.

API width: Additional system call entrypoints (or additional options to existing system calls) may be required to represent these intermediate states, effectively widening the kernel's API. For example, in the Fluke API, there are five system calls that are rarely called directly from user-mode programs, and are instead usually only used as "restart points" for interrupted kernel operations. However, we have found in practice that although these seldom-used entrypoints are mandated by the fully-interruptible API design, they are also directly useful to some applications; there are no Fluke entrypoints whose purpose is solely to provide a pure interrupt-model API.

Actual Cause of Exception	Cost to Remedy	Cost to Rollback
Client-side soft page fault	18.9	none
Client-side hard page fault	118	2.2
Server-side soft page fault	29.3	2.5
Server-side hard page fault	135	6.8

Table 3: Breakdown of restart costs in microseconds for possible kernel-internal exceptions during a reliable IPC transfer, the area of the kernel with the most internal synchronization (specifically, `ipc_client_connect_send_over_receive`). "Actual Cause" describes the reason the exception was raised: either a "soft" page fault (one for which the kernel can derive a page table entry based on an entry higher in the memory mapping hierarchy) or a "hard" page fault (requiring an RPC to a user-level memory manager) in either the client or server side of the IPC. "Cost to Rollback" is roughly the amount of work thrown away and redone, while "Cost to Remedy" approximates the amount of work needed to service the fault. Results were obtained on a 200-Mhz Pentium Pro with the Fluke kernel configured using a process model without kernel thread preemption.

Thread state size: Additional user-visible thread state may be required. For example, in Fluke on the x86, due to the shortage of processor registers, two 32-bit "pseudo-registers" implemented by the kernel are included in the user-visible thread state frame to hold intermediate IPC state. These pseudo-registers add a little more complexity to the API, but they never need to be accessed directly by user code except when saving and restoring thread state, so they do not in practice cause a performance burden.

Overhead from Restarting Operations: During some system calls, various events can cause the thread's state to be rolled back, requiring a certain amount of work to be re-done later. Our measurements, summarized in Table 3, show this not to be a significant cost. Application threads rarely access each other's state (e.g., only during the occasional checkpoint or migration), so although it is important for this to be possible, it is not the common case. The only other situation in which threads are rolled back is when an exception such as a page fault occurs, and in such cases, the time required to handle the exception invariably dwarfs the time spent re-executing a small piece of system call code later.

Architectural bias: Certain older processor architectures make it impossible for the kernel to provide cor-

rect and prompt state exportability, because the processor itself does not do so. For example, the Motorola 68020/030 saved state frame includes some undocumented fields whose contents must be kept unmodified by the kernel; these fields cannot safely be made accessible and modifiable by user-mode software, and therefore a thread's state can never be fully exportable when certain floating-point operations are in progress. However, most other architectures, including the x86 and even other 68000-class processors, such as the 68040, do not have this problem.

In practice, none of these disadvantages has caused us significant problems in comparison to the benefits of correct, prompt state exportability.

5 Kernel Execution Models

We now return to the issue of the execution model used in a kernel's *implementation*. While there is typically a strong correlation between a kernel's API and its internal execution model, in many ways these issues are independent. In this section we report our experiments with Fluke and, previously, with Mach, that demonstrate the following findings.

Exported API: A process-model kernel can easily implement either style of API, but an interrupt-model kernel has a strong "preference" for an atomic API.

Preemptibility: It is easier to make a process-model kernel preemptible, regardless of the API it exports; however, it is easy to make interrupt-model kernels partly preemptible by adding preemption points.

Performance: Depending on the application running on the kernel, either a process-model or interrupt-model kernel can be faster, but not by much. In terms of preemption latency, an interrupt-model kernel can perform as well as an equivalently configured process-model kernel, but a fully-preemptible process-model kernel provides the lowest latency.

Memory use: Naturally, process-model kernels use more memory because of the larger number of kernel stacks in the system; on the other hand, the size of kernel stacks sometimes can be reduced to minimize this disadvantage.

Architectural bias: Some CISC architectures that insist on providing automatic stack handling, such as the x86, are fundamentally biased towards the process model, whereas most RISC architectures support both models equally well.

Legacy code: Since most existing, robust, easily available OS code, such as device drivers and file systems, is written for the process model, it is easiest to use this legacy code in process-model kernels. However, it is also possible to use this code in interrupt-model kernels with a slight performance penalty.

```
msg_send_rcv(msg, option,
             send_size, rcv_size, ...) {
        rc = msg_send(msg, option,
                      send_size, ...);
        if (rc != SUCCESS)
                return rc;

        rc = msg_rcv(msg, option, rcv_size, ...);
        return rc;
}
```

Figure 2: An example IPC send and receive path in a process model kernel. Any waiting or fault handling during the operation must keep the kernel stack bound to the current thread.

The following sections discuss these issues in detail and provide concrete measurement results where possible.

5.1 Exported API

In kernels with medium-to-high level API's, one of the most common objections to the interrupt-based execution model is that it requires the kernel to maintain explicit continuations. Our observation is that continuations are not a fundamental property of an interrupt-model kernel, but instead are the symptom of the mismatch between the kernel's API and its implementation. In brief, an interrupt-model kernel only requires continuations when implementing a conventional API; when an interrupt-model kernel serves an atomic API, *explicit* user-visible register state of a thread acts as the "continuation."

Continuations

To illustrate this difference, consider the IPC pseudocode fragments in Figures 2, 3, and 4. The first shows a very simplified version of a combined IPC message send-and-receive system call similar to the `mach_msg_trap` system call inside the original process-model Mach 3.0 kernel. The code first calls a subroutine to send a message; if that succeeds, it then calls a second routine to receive a message. If an error occurs in either stage, the entire operation is aborted and the system call finishes by passing a return code back to the user-mode caller. This structure implies that any exceptional conditions that occur along the IPC path that should not cause the operation to be completely aborted, such as the need to wait for an incoming message or service a page fault, must be handled completely within these subroutines by blocking the current thread while retaining its kernel stack. Once the `msg_send_rcv` call returns, the system call is complete.

Figure 3 shows pseudocode for the same IPC path modified to use a partial interrupt-style execution environment, as was done by Draves in the Mach 3.0 continuations work [10, 11]. The first stage of the operation, `msg_send`, is expected to retain the current kernel stack, as above; any page faults or other temporary conditions

```
msg_send_rcv(msg, option,
             send_size, rcv_size, ...) {
    rc = msg_send(msg, option,
                  send_size, ...);
    if (rc != SUCCESS)
            return rc;

    cur_thread->continuation.msg = msg;
    cur_thread->continuation.option = option;
    cur_thread->continuation.rcv_size = rcv_size;
    ...

    rc = msg_rcv(msg, option, rcv_size, ...,
                 msg_rcv_continue);
    return rc;
}

msg_rcv_continue(cur_thread) {
    msg = cur_thread->continuation.msg;
    option = cur_thread->continuation.option;
    rcv_size = cur_thread->continuation.rcv_size;
    ...

    rc = msg_rcv(msg, option, rcv_size, ...,
                 msg_rcv_continue);
    return rc;
}
```

Figure 3: Example interrupt model IPC send and receive path. State defining the "middle" of the send-receive is saved away by the kernel after `msg_send` in the case that the `msg_rcv` is interrupted. Special code, `msg_rcv_continue`, is needed to handle restart from a continuation.

during this stage must be handled in process-model fashion, without discarding the stack. However, in the common case where the subsequent receive operation must wait for an incoming message, the `msg_rcv` function can discard the kernel stack while waiting. When the wait is satisfied or interrupted, the thread will be given a new kernel stack and the `msg_rcv_continue` function will be called to finish processing the `msg_send_rcv` system call. The original parameters to the system call must be saved explicitly in a continuation structure in the current thread, since they are not retained on the kernel stack.

Note that although this modification partly changes the system call to have an interrupt model *implementation*, it still retains its conventional *API semantics* as seen by user code. For example, if another thread attempts to examine this thread's state while it is waiting continuation-style for an incoming message, the other thread will either have to wait until the system call is completed, or the system call will have to be aborted, causing loss of state.[4] This is because the thread's continuation structure,

[4]In this particular situation in Mach, the `mach_msg_trap` operation gets aborted with a special return code; standard library user-mode code can detect this situation and manually restart the IPC. However, there are many other situations, such as page faults occurring along the IPC path while copying data, which, if aborted, cannot be reliably restarted in this way.

```
msg_send_rcv(cur_thread) {
        rc = msg_send(cur_thread);
        if (rc != SUCCESS)
                return rc;

        set_pc(cur_thread, msg_rcv_entry);
        rc = msg_rcv(cur_thread);

        if (rc != SUCCESS)
                return rc;
        return SUCCESS;
}
```

Figure 4: Example IPC send and receive path for a kernel exporting an atomic API. The `set_pc` operation effectively serves the same purpose as saving a continuation, using the user-visible register state as the storage area for the continuation. Exposing this state to user mode as part of the API provides the benefits of a purely atomic API and eliminates much of the traditional complexity of continuations. The kernel never needs to save parameters or other continuation state on entry because it is already in the thread's user-mode register state.

including the continuation function pointer itself (pointing to `msg_rcv_continue`), is part of the thread's logical state but is inaccessible to user code.

Interrupt-Model Kernels Without Continuations

Finally, contrast these two examples with corresponding code in the style used throughout the Fluke kernel, shown in Figure 4. Although this code at first appears very similar to the code in Figure 2, it has several fundamental differences. First, system call parameters are passed in registers rather than on the user stack. The system call entry and exit code saves the appropriate registers into the thread's control block in a standard format, where the kernel can read and update the parameters. Second, since the system call parameters are stored in the register save area of the thread's control block, no unique, per-system call continuation state is needed. Third, when an internal system call handler returns a nonzero result code, the system call exit layer does *not* simply complete the system call and pass this result code back to the user. Return values in the kernel are only used for *kernel-internal* exception processing; results intended to be seen by user code are returned by modifying the thread's saved user-mode register state. Finally, if the `msg_send` stage in `msg_send_rcv` completes successfully, the kernel updates the user-mode program counter to point to the user-mode system call entrypoint for `msg_rcv` before proceeding with the `msg_rcv` stage. Thus, if the `msg_rcv` must wait or encounters a page fault, it can simply return an appropriate (kernel-internal) result code. The thread's user-mode register state will be left so that when normal processing is eventually resumed, the `msg_rcv` system call will automatically be invoked with the appropriate parameters to finish the IPC

operation.

The upshot of this is that in the Fluke kernel, the thread's explicit user-mode register state acts as the "continuation," allowing the kernel stack to be thrown away or reused by another thread if the system call must wait or handle an exception. Since a thread's user-mode register state is *explicit* and fully visible to user-mode code, it can be exported at any time to other threads, thereby providing the promptness and correctness properties required by the atomic API. Furthermore, this atomic API in turn simplifies the interrupt model kernel implementation to the point of being almost as simple and clear as the original process model code in Figure 2.

5.2 Preemptibility

Although the use of an atomic API greatly reduces the kernel complexity and the burden traditionally associated with interrupt-model kernels, there are other relevant factors as well, such as kernel preemptibility. Low preemption latency is a desirable kernel characteristic, and is critical in real-time systems and in microkernels such as L3 [23] and VSTa [33] that dispatch hardware interrupts to device drivers running as ordinary threads (in which case preemption latency effectively becomes interrupt-handling latency). Since preemption can generally occur at any time while running in user mode, it is the kernel itself that causes preemption latencies that are greater than the hardware minimum.

In a process-model kernel that already supports multiprocessors, it is often relatively straightforward to make most of the kernel preemptible by changing spin locks into blocking locks (e.g., mutexes). Of course, a certain core component of the kernel, which implements scheduling and preemption itself, must still remain non-preemptible. Implementing kernel preemptibility in this manner fundamentally relies on kernel stacks being retained by preempted threads, so it clearly would not work in a pure interrupt-model kernel. The Fluke kernel can be configured to support this form of kernel preemptibility in the process model.

Even in an interrupt model kernel, important parts of the kernel can often be made preemptible as long as preemption is carefully controlled. For example, in microkernels that rely heavily on IPC, many long-running kernel operations tend to be IPCs that copy data from one process to another. It is relatively easy to support partial preemptibility in a kernel by introducing *preemption points* in select locations, such as on the data copy path. Besides supporting full kernel preemptibility in the process model, the Fluke kernel also supports partial preemptibility in both execution models. QNX [19] is an example of another existing interrupt model kernel whose IPC path is made preemptible in this fashion.

Configuration	Description
Process NP	Process model with no kernel preemption. Requires no kernel-internal locking. Comparable to a uniprocessor Unix system.
Process PP	Process model with "partial" kernel preemption. A single explicit preemption point is added on the IPC data copy path, checked after every 8k of data is transferred. Requires no kernel locking.
Process FP	Process model with full kernel preemption. Requires blocking mutex locks for kernel locking.
Interrupt NP	Interrupt model with no kernel preemption. Requires no kernel locking.
Interrupt PP	Interrupt model with partial preemption. Uses the same IPC preemption point as in Process PP. Requires no kernel locking.

Table 4: Labels and characteristics for the different Fluke kernel configurations used in test results.

5.3 Performance

The Fluke kernel supports a variety of build-time configuration options that control the execution model of the kernel; by comparing different configurations of the same kernel, we can analyze the properties of these different execution models. We explore kernel configurations along two axes: interrupt versus process model and full versus partial (explicit preemption points) versus no preemption. Since full kernel preemptibility requires the ability to block within the kernel and is therefore incompatible with the interrupt model, there are five possible configurations, summarized in Table 4.

Table 5 shows the relative performance of three applications on the Fluke kernel under various kernel configurations. For each application, the execution times for all kernel configurations are normalized to the execution time of that application on the "base" configuration: process model with no kernel preemption. For calibration, the raw execution time is given for the base configuration. The non-fully-preemptible kernels were run both with and without partial preemption support on the IPC path. All tests were run on a 200MHz Pentium Pro PC with 256KB L2 cache and 64MB of memory. The applications measured are:

Flukeperf: A series of tests to time various synchronization and IPC primitives. It performs a large number of kernel calls and context switches.

Memtest: Accesses 16MB of memory one byte at a time sequentially. Memtest runs under a memory manager which allocates memory on demand, exercising kernel fault handling and the exception IPC facility.

Gcc: Compile a single .c file. This test include running the front end, the C preprocessor, C compiler, assembler

Configuration	memtest	flukeperf	gcc
Process NP	1.00 (2884*ms*)	1.00 (7120*ms*)	1.00 (7150*ms*)
Process PP	1.00	1.01	1.03
Process FP	1.11	1.20	1.05
Interrupt NP	1.00	0.94	1.03
Interrupt PP	1.00	0.94	1.03

Table 5: Performance of three applications on various configurations of Fluke kernel. Execution time is normalized to the performance of the process-model kernel without kernel preemption (Process NP) for which absolute times are also given.

and linker to produce a runnable Fluke binary.

As expected, performance of a fully-preemptible kernel is somewhat worse than the other configurations due to the need for kernel locking. The extent of the degradation varies from 20% for the kernel-intensive *flukeperf* test to only 5% for the more user-mode oriented *gcc*. Otherwise, the interrupt and process model kernels are nearly identical in performance except for the *flukeperf* case. In *flukeperf* we are seeing a positive effect of an interrupt model kernel implementation. Since a thread will restart an operation after blocking rather than resuming from where it slept in the kernel, there is no need to save the thread's kernel-mode register state on a context switch. In Fluke this translates to eliminating six 32-bit memory reads and writes on every context switch.

Because the applications shown are all single-threaded, the results in Table 5 do not realistically reflect the impact of preemption. To measure the effect of the execution model on preemption latency, we introduce a second, high-priority kernel thread which is scheduled every millisecond, and record its observed preemption latencies during a run of *flukeperf*. The *flukeperf* application is used because it performs a number of large, long running IPC operations ideal for inducing preemption latencies.

Table 6 summarizes the experiment. The first two columns are the average and maximum observed latency in microseconds. The last two columns of the table show the number of times the high-priority kernel thread ran over the course of the application and the number of times it could not be scheduled because it was still running or queued from the previous interval. As expected, the fully-preemptible (FP) kernel permits much smaller and predictable latencies and allowed the high-priority thread to run without missing an event. The non-preemptible (NP) kernel configuration exhibits highly variable latency for both the process and interrupt model causing a large number of missed events. Though we implement only a single explicit preemption point on the IPC data copy path, the partial preemption (PP) configuration fares well on this benchmark. This result is not surprising given that the benchmark performs a number of large IPC operations, but it illustrates that a few well-placed preemption points can greatly reduce preemption latency in an otherwise nonpreemptible kernel.

Configuration	flukeperf			
	latency		schedules	
	avg	max	run	miss
Process NP	28.9	7430	7594	132
Process PP	18.0	1200	7805	5
Process FP	5.14	19.6	9212	0
Interrupt NP	30.4	7356	7348	141
Interrupt PP	18.7	1272	7531	7

Table 6: Effect of execution model on preemption latency. We measure the average and maximum time (μs) required for a periodic high-priority kernel thread to start running after being scheduled, while competing with lower-priority application threads. Also shown is the number of times the kernel thread runs during the lifetime of the application and the number of times it failed to complete before the next scheduling interval.

5.4 Memory Use

One of the perceived benefits of the interrupt model is the memory saved by having only one kernel stack per processor rather than one per thread. For example, Mach's average per-thread kernel memory overhead was reduced by 85% when the kernel was changed to use a partial interrupt model [10, 11]. Of course, the overall memory used in a system for thread management overhead depends not only on whether each thread has its own kernel stack, but also on how big these kernel stacks are and how many threads are generally used in a realistic system.

To provide an idea of how these factors add up in practice, we show in Table 7 memory usage measurements gathered from a number of different systems and configurations. The Mach figures are as reported in [10]: the process-model numbers are from MK32, an earlier version of the Mach kernel, whereas the interrupt-model numbers are from MK40. The L3 figures are as reported in [24]. For Fluke, we show three different rows: two for the process model using two different stack sizes, and one for the interrupt model.

The two process-model stack sizes for Fluke bear special attention. The smaller 1K stack size is sufficient only in the "production" kernel configuration which leaves out various kernel debugging features, and only when the device drivers do not run on these kernel stacks. (Fluke's device drivers were "borrowed" [14] from legacy systems and require a 4K stack.)

To summarize these results, although it is true that interrupt-model kernels most effectively minimize kernel thread memory use, at least for modest numbers of active threads, much of this reduction can also be achieved in process-model kernels simply by structuring the ker-

System	Execution Model	TCB Size	Stack Size	Total Size
FreeBSD	Process	2132	6700	8832
Linux	Process	2395	4096	6491
Mach	Process	452	4022	4474
Mach	Interrupt	690	—	690
L3	Process	1024		1024
Fluke	Process	4096		4096
Fluke	Process	1024		1024
Fluke	Interrupt	300	—	300

Table 7: Memory overhead in bytes due to thread/process management in various existing systems and execution models.

nel to avoid excessive stack requirements. At least on the x86 architecture, as long as the thread management overhead is about 1K or less per thread, there appears to be no great difference between the two models for modest numbers of threads. However, real production systems may need larger stacks and also may want them to be a multiple of the page size in order to use a "red zone." These results should apply to other architectures, although the basic sizes may be scaled by an architecture-specific factor. For all but power-constrained systems, the memory differences are probably in the noise.

5.5 Architectural Bias

Besides the more fundamental advantages and disadvantages of each model as discussed above, in some cases there are advantages to one model artificially caused by the design of the underlying processor architecture. In particular, traditional CISC architectures, such as the x86 and 680x0, tend to be biased somewhat toward the process model and make the kernel programmer jump through various hoops to write an interrupt-model kernel. With a few exceptions, more recent RISC architectures tend to be fairly unbiased, allowing either model to be implemented with equal ease and efficiency.

Unsurprisingly, the architectural property that causes this bias is the presence of automatic stack management and stack switching performed by the processor. For example, when the processor enters supervisor mode on the x86, it automatically loads the new supervisor-mode stack pointer, and then pushes the user-mode stack pointer, instruction pointer (program counter), and possibly several other registers onto this supervisor-mode stack. Thus, the processor automatically *assumes* that the kernel stack is associated with the current thread. To build an interrupt-model kernel on such a "process-model architecture," the kernel must either copy this data on kernel entry from the per-processor stack to the appropriate thread control block, or it must keep a separate, "minimal" process-model stack as part of each thread control block, where the processor automatically saves the thread's state on kernel entry before kernel code manually switches to the "real" kernel stack. Fluke in its interrupt-model configuration uses the former technique, while Mach uses the latter.

Most RISC processors, on the other hand, including the MIPS, PA-RISC, and PowerPC, use "shadow registers" for exception and interrupt handling rather than explicitly supporting stack switching in hardware. When an interrupt or exception occurs, the processor merely saves off the original user-mode registers in special one-of-a-kind shadow registers, and then disables further interrupts until they are explicitly re-enabled by software. If the OS wants to support nested exceptions or interrupts, it must then store these registers on the stack itself; it is generally just as easy for the OS to save them on a per-processor interrupt-model stack as it is to save them on a per-thread process-model stack. A notable exception among RISC processors is the SPARC, with its stack-based register window feature.

To examine the effect of architectural bias on the x86, we compared the performance of the interrupt and process-model Fluke kernels in otherwise completely equivalent configurations (using no kernel preemption). On a 100MHz Pentium CPU, the additional trap and system call overhead introduced in the interrupt-model kernel by moving the saved state from the kernel stack to the thread structure on entry, and back again on exit, amounts to about six cycles (60ns). In contrast, the minimal hardware-mandated cost of entering and leaving supervisor mode is about 70 cycles on this processor. Therefore, even for the fastest possible system call the interrupt-model overhead is less than 10%, and for realistic system calls is in the noise. We conclude that although this architectural bias is a significant factor in terms of programming convenience, and may be important if it is necessary to "squeeze every last cycle" out of a critical path, it is not a major performance concern in general.

5.6 Legacy Code

One of the most important practical concerns with an interrupt-based kernel execution model is that it appears to be impossible to use pre-existing legacy code, borrowed from process-model systems such as BSD or Linux, in an interrupt-model kernel, such as the exokernels and the CacheKernel. For example, especially on the x86 architecture, it is impractical for a small programming team to write device drivers for any significant fraction of the commonly available PC hardware; they must either borrow drivers from existing systems, or support only a bare minimum set of hardware configurations. The situation is similar, though not as severe, for other types of legacy code such as file systems or TCP/IP protocol stacks.

There are a number of approaches to incorporating process-model legacy code into interrupt-model kernels.

For example, if kernel threads are available (threads that run in the kernel but are otherwise ordinary process-model threads), process-model code can be run on these threads when necessary. This is the method Minix [30] uses to run device driver code. Unfortunately, kernel threads can be difficult to implement in interrupt-model kernels, and can introduce additional overhead on the kernel entry/exit paths, especially on architectures with the process-model bias discussed above. This is because such processors behave differently in a trap or interrupt depending on whether the interrupted code was in user or supervisor mode [21]; therefore each trap or interrupt handler in the kernel must now determine whether the interrupted code was a user thread, a process-model kernel thread, or the interrupt-model "core" kernel itself, and react appropriately in each case. In addition, the process-model stacks of kernel threads on these architectures can't easily be pageable or dynamically growable, because the processor depends on always being able to push saved state onto the kernel stack if a trap occurs. Ironically, on RISC processors that have no bias towards the process model, it is much easier to implement process-model kernel threads in an interrupt-model kernel.

As an alternative to supporting kernel threads, the kernel can instead use only a *partial* interrupt model, in which kernel stacks are usually handed off to the next thread when a thread blocks, but can be retained while executing process-model code. This is the method that Mach with continuations [11] uses. Unfortunately, this approach brings with it a whole new set of complexities and inefficiencies, largely caused by the need to manage kernel stacks as first-class kernel objects independent of and separable from both threads and processors.

The Fluke kernel uses a different approach, which keeps the "core" interrupt-model kernel simple and uncluttered while effectively supporting something almost equivalent to kernel threads. Basically, the idea is to run process-model "kernel" threads in user mode but in the kernel's address space. In other words, these threads run in the processor's unprivileged execution mode, and thus run on their own process-model user stacks separate from the kernel's interrupt-model stack; however, the address translation hardware is set up so that while these threads are executing, their view of memory is effectively the same as it is for the "core" interrupt-model kernel itself. This design allows the core kernel to treat these process-level activities just like all other user-level activities running in separate address spaces, except that this particular address space is set up a bit differently.

There are three main issues with this approach. The first is that these user-level pseudo-kernel threads may need to perform *privileged operations* occasionally, for example to enable or disable interrupts or access device registers. In the x86 this isn't a problem because user-level threads can be given direct access to these facilities simply by setting some processor flag bits associated with those threads; however, on other architectures these operations may need to be "exported" from the core kernel as pseudo-system calls only available to these special pseudo-kernel threads. Second, these user-level activities may need to synchronize and share data structures with the core kernel to perform operations such as allocating kernel memory or installing interrupt handlers; since these threads are treated as normal user-mode threads, they are probably fully preemptible and do not share the same constrained execution environment or synchronization primitives as the core kernel uses. Again, a straightforward solution, which is what Fluke does, is to "export" the necessary facilities through a special system call that allows these special threads to temporarily jump into supervisor mode and the kernel's execution environment, perform some arbitrary (nonblocking) activity, and then return to user mode. The third issue is the *cost* of performing this extra mode switching; however, our experience indicates that this cost is negligible. Finally, note that the memory management hardware on some processors, particularly the MIPS architecture, does not support this technique; however, at least on MIPS there is no compelling need for it either because the processor does not have the traditional CISC-architecture bias toward a particular kernel stack management scheme.

6 Conclusion

In this paper, we have explored in depth the differences between the interrupt and process models and presented a number of ideas, insights, and results. Our Fluke kernel demonstrates that the need for the kernel to manually save state in *continuations* is not a fundamental property of the interrupt model, but instead is a symptom of a mismatch between the kernel's implementation and its API. Our kernel exports a purely "atomic" API, in which all kernel operations are fully interruptible and restartable; this property has important benefits for fault-tolerance and for applications such as user-mode process migration, checkpointing, and garbage collection, and eliminates the need for interrupt-model kernels to manually save and restore continuations. Using our configurable kernel which supports both the interrupt-based and process-based execution models, we have made a controlled comparison between the two execution models. As expected, the interrupt-model kernel requires less per-thread memory. Although a null system call entails a 5–10% higher overhead on an interrupt-model kernel due to a built-in bias toward the process model in common processor architectures such as the x86, the interrupt-model kernel exhibits a modest performance advantage in some cases. However, the interrupt model can incur vastly higher preemp-

tion latencies unless care is taken to insert explicit preemption points on critical kernel paths. Our conclusion is that it is highly desirable for a kernel to present an atomic API such as Fluke's, but that for the kernel's internal execution model, either implementation model is reasonable.

Acknowledgements

We are grateful to Alan Bawden and Mootaz Elnozahy for interesting and enlightening discussion concerning these interface issues and their implications for reliability, to Kevin Van Maren for elucidating and writing up notes on other aspects of the kernel's execution model, to the anonymous reviewers, John Carter, and Dave Andersen for their extensive and helpful comments, to Linus Kamb for help with the use of certain system calls, and to Eric Eide for last minute expert formatting help.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of the Summer 1986 USENIX Conf.*, pages 93–112, June 1986.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, Feb. 1992.

[3] J. F. Bartlett. A Non Stop Kernel. In *Proc. of the 8th ACM Symposium on Operating Systems Principles*, pages 22–29, Dec. 1981.

[4] A. Bawden. PCLSRing: Keeping Process State Modular. Unpublished report. ftp://ftp.ai.mit.edu/pub/alan/pclsr.memo, 1989.

[5] A. Bawden. Personal Communication, Aug. 1998.

[6] A. C. Bomberger and N. Hardy. The KeyKOS Nanokernel Architecture. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Apr. 1992.

[7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pages 152–164, Oct. 1991.

[8] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, Mar. 1988.

[9] D. R. Cheriton and K. J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 179–193. USENIX Assoc., Nov. 1994.

[10] R. P. Draves. *Control Transfer in Operating System Kernels*. PhD thesis, Carnegie Mellon University, May 1994.

[11] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, Asilomar, CA, Oct. 1991.

[12] D. Eastlake, R. Greenblatt, J. Holloway, T. Knight, and S. Nelson. ITS 1.5 Reference Manual. Memo 161a, MIT AI Lab, July 1969.

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 251–266, Dec. 1995.

[14] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for OS and Language Research. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 38–51, St. Malo, France, Oct. 1997.

[15] B. Ford, M. Hibler, and Flux Project Members. Fluke: Flexible μ-kernel Environment (draft documents). University of Utah. Postscript and HTML available under http://www.cs.utah.edu/-projects/flux/fluke/, 1996.

[16] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 137–151. USENIX Assoc., Oct. 1996.

[17] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan. Recovery Management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, Feb. 1988.

[18] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1989.

[19] D. Hildebrand. An Architectural Overview of QNX. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, Apr. 1992.

[20] Institute of Electrical and Electronics Engineers, Inc. *Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, 1994. Std 1003.1-1990.

[21] Intel Corp. *Pentium Processor User's Manual*, volume 3. Intel, 1993.

[22] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. B. no, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Janotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 52–65, St. Malo, France, Oct. 1997.

[23] J. Liedtke. A Persistent System in Real Use – Experiences of the First 13 Years. In *Proc. of the Third International Workshop on Object Orientation in Operating Systems*, pages 2–11, Dec. 1993.

[24] J. Liedtke. A Short Note on Small Virtually-Addressed Control Blocks. *Operating Systems Review*, 29(3):31–34, July 1995.

[25] P. R. McJones and G. F. Swart. Evolving the UNIX System Interface to Support Multithreaded Programs. In *Proceedings of the Winter 1989 USENIX Technical Conference*, pages 393–404, San Diego, CA, Feb. 1989. USENIX.

[26] S. J. Mullender. Process Management in a Distributed Operating System. In J. Nehmer, editor, *Experiences with Distributed Systems*, volume 309 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[27] Open Software Foundation and Carnegie Mellon Univ. *OSF MACH Kernel Principles*, 1993.

[28] M. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, Feb. 1990.

[29] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, Oct. 1996. USENIX Assoc.

[30] A. S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[31] P. Tullmann, J. Lepreau, B. Ford, and M. Hibler. User-level Checkpointing Through Exportable Kernel State. In *Proc. Fifth International Workshop on Object Orientation in Operating Systems*, pages 85–88, Seattle, WA, Oct. 1996. IEEE.

[32] V-System Development Group. *V-System 6.0 Reference Manual*. Computer Systems Laboratory, Stanford University, May 1986.

[33] A. Valencia. An Overview of the VSTa Microkernel. http://-www.igcom.net/~jeske/VSTa/vsta_intro.html.

Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels[1]

Ariel Tamches and Barton P. Miller
Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685
{tamches,bart}@cs.wisc.edu

Abstract

We have developed a technology, fine-grained dynamic instrumentation of commodity kernels, which can splice (insert) dynamically generated code before almost any machine code instruction of a completely unmodified running commodity operating system kernel. This technology is well-suited to performance profiling, debugging, code coverage, security auditing, runtime code optimizations, and kernel extensions. We have designed and implemented a tool called KernInst that performs dynamic instrumentation on a stock production Solaris kernel running on an UltraSPARC. On top of KernInst, we have implemented a kernel performance profiling tool, and used it to understand kernel and application performance under a Web proxy server workload. We used this information to make two changes (one to the kernel, one to the proxy) that cumulatively reduce the percentage of elapsed time that the proxy spends opening disk cache files from 40% to 7%.

1 Introduction

Operating system kernels are complex entities whose internals often are difficult to understand, much less measure and optimize. Recently, extensible kernels, such as SPIN, Exokernel, and VINO, have been designed to allow applications to extend functionality and specify kernel policies [4,6,17]. Synthetix allows specialized versions of certain kernel functions to be installed at runtime, providing dynamic optimization [16]. A design has even been proposed for a self-measuring and self-adapting extensible kernel [18]. All of the above work has been performed on customized kernels, so it is difficult to evaluate or use with real-world programs and workloads. This paper introduces *fine-grained dynamic kernel instrumentation*, a low-level technology that allows arbitrary code to be *spliced* (inserted) at almost any kernel machine code location during runtime. Dynamic kernel instrumentation allows runtime measurements, optimizations, and extensions to be performed on unmodified commodity kernels. In this paper, we provide a motivation for fine-grained dynamic kernel instrumentation and describe how to dynamically instrument an unmodified commodity kernel. We show a kernel profiler that, using dynamic instrumentation, provides a two-way benefit: insight into both kernel and application performance. We show how this information was used to optimize a web proxy server. We also discuss safety and security issues introduced by fine-grained dynamic kernel instrumentation.

Dynamic instrumentation supports monitoring functionality, such as debugging and profiling, alongside mechanisms for extensibility and adaptability, *in a single infrastructure*. Kernels become evolving entities, able to measure and adapt themselves to accommodate real-world runtime usage patterns.

The main contribution of our work is the design and implementation of a fine-grained splicing mechanism for a stock commodity kernel. The contents of the inserted code—whether performance profiling annotations, optimized versions of functions, or process-specific kernel extensions—are orthogonal to the issue of how to splice it into a commodity kernel.

We have implemented *KernInst*, an instrumentation tool for the Solaris kernel. Its main features are:

Fully dynamic. KernInst is loaded and instruments a running kernel, without any need to recompile, reboot, or even pause the kernel.

Fine-grained. *Instrumentation points* (locations where code can be spliced) can be almost any machine code instruction in the kernel. This contrasts with kernels that allow coarser-grained code changes, such as at function granularity (e.g., VINO [17] and Synthetix [16]), or only allow entire kernel modules to be changed (which many commodity kernels allow).

Runs on a commodity kernel. This allows us to immediately run real-world programs.

Runs on an *unmodified* kernel. Any UltraSPARC system running Solaris can immediately use KernInst.

1. This work is supported in part by Department of Energy Grant DE-FG02-93ER25176, NSF grants EIA-9870684 and CDA-9623632, and DARPA contract N66001-97-C-8532. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

2 Applications of Dynamic Kernel Instrumentation

This section describes several applications for fine-grained dynamic instrumentation of unmodified commodity kernels.

Performance profilers can use dynamic instrumentation to insert performance-gathering code annotations, such as incrementing a counter at the start of a function or basic block. Annotation code also can start and stop timers or access hardware performance counters. More complex code sequences and control structures can be used to predicate measurements on kernel state, such as the currently running process. More detail on using dynamic kernel instrumentation for performance profiling is presented in Section 4.

Dynamic instrumentation can also be used for kernel tracing, by splicing logging code at the desired kernel code locations during runtime. When the desired trace is collected, kernel code can be restored to its original contents, so overhead is incurred only when tracing is desired. This contrasts with a static kernel instrumentation system (such as a binary rewriter), which would insert code lasting for the entire run of the kernel.

Code coverage can be measured by splicing code that sets a flag (one per basic block) indicating that code has been reached. Instrumentation for a basic block can be removed as soon as the flag is set; thus, the overhead of code coverage actually *decreases* over time. Basic block coverage demonstrates the need for instrumentation to be fine-grained.

Kernel debuggers can be implemented using fine-grained dynamic instrumentation. Breakpoints can be inserted at any machine code instruction by splicing code that displays kernel state and (optionally) pauses the executing thread and informs the debugger. Conditional breakpoints are easily accommodated by predicating the breakpoint with the appropriate condition.

Security auditing annotations can be installed using dynamic instrumentation. Solaris can audit thread creation and deletion, file system pathname lookups, file system vnode creation, successful and unsuccessful credential checks for super-user access, process forks, core dumps, stream device operations, file opens, closes, and chdirs, and many more. However, auditing code is turned off by default; turning it on requires a kernel recompile and reboot. With dynamic instrumentation, the auditing package can be distributed as an independent kernel add-on, and installed onto a running system. This requires a fine-grained splicing mechanism, since auditing checks often take place in the middle of kernel functions.

Dynamic instrumentation enables automated runtime code optimization, based on performance feedback gathered by dynamic profiling annotations. One example is function specialization [16] on an input parameter. A function can be dynamically instrumented to collect a histogram of the desired parameter, which is later examined for a frequent value. Annotation code is then removed, and a specialized version of the function's machine code is generated, with constant propagation applied to the specialized parameter. The function then has the following code spliced at its entry: "if the input parameter equals the common value, then jump to the optimized version; else, fall through to the original version". A further optimization can sometimes bypass this check: sites where the function is called can be examined for an actual parameter that always equals the specialized value. If so, the call site is altered to directly call the optimized version of the function.

Moving seldom-executed basic blocks out of line to improve instruction-cache behavior [13] can be performed using fine-grained dynamic instrumentation. A function's entry and exit point(s) can be annotated to measure the number of icache misses it incurs. If the value is high, the function's basic blocks can be instrumented to count execution frequency. An optimized version of the function, with infrequently executed blocks moved out of line, is then installed by splicing in code at the entry of the original function to unconditionally jump to the optimized version. As with parameter specialization, the extra jump overhead can often be eliminated by altering sites where the function is called.

Dynamic kernel instrumentation also may be used to change kernel functionality, such as installing a process-specific version of a kernel policy. Extensible operating systems that download process-specific code into the kernel for improved performance [3,6,14,17] perform this kind of adaptation. Dynamic instrumentation can easily provide a similar operation in a commodity kernel by splicing the following code at a desired kernel function: "if currently executing process id equals *customized pid* then jump to customized version in the patch area; else, fall through".

Applications using dynamic kernel instrumentation have varying requirements for kernel source code. KernInst provides the kernel's runtime symbol table (function names and their starting locations in memory) and a control flow graph of basic blocks at the machine code level. With this information, applications can readily identify the machine code locations of function entries, exits, and function calls. A benefit of working with machine code is that the effect of compiler optimizations, which can reorder and remove code, are visible. Certain applications require more information about the kernel. A kernel developer using dynamic kernel instrumentation to trace a specific source code line needs the

compiler's debugging line number information to map line numbers into kernel addresses. (Fortunately, kernel developers are likely to have access to this information.) Other applications may require knowing only the names of specific kernel functions. An application wanting to profile file system pathname-to-vnode translation in Solaris needs to know that the relevant function is lookuppn. (End users of such an application do *not* need to know this.) Other applications will work solely with the information KernInst provides; for example, an optimizer that moves seldom-executed basic blocks out of line at run-time works at the machine code level.

3 Mechanisms

Dynamic kernel instrumentation is the process of splicing dynamically generated code sequences into specified points in the kernel code space. Splicing overwrites the machine code instruction at an *instrumentation point* with a jump to patch code. KernInst is fine-grained; instrumentation points can be almost any machine code instruction in the kernel (we will discuss the exceptions in Section 3.3 and Section 3.4.1). The code patch contains the dynamically generated code being inserted, the overwritten instruction, and a jump back to the instruction following the instrumentation point. The net effect of splicing is to insert dynamically generated code before a given kernel machine code instruction.

An important feature of our dynamic instrumentation is that splicing is independent of code generation. KernInst can splice machine code that has been created from code generation packages such as VCODE [7], an interpreter performing just-in-time compilation, or from precompiled position-independent code.

The structure of the KernInst system is shown in

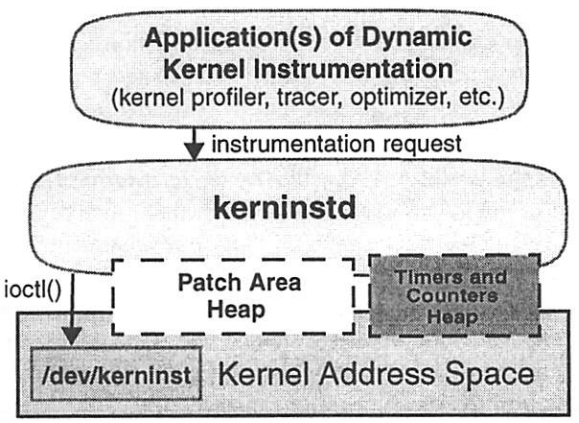


Figure 1: KernInst System Architecture
Kerninstd acts as an instrumentation server, performing kernel instrumentation requests that arrive from applications.

Figure 1. Applications that wish to instrument the kernel interact with *kerninstd*, a user-level daemon. There is also a small runtime-loaded KernInst driver */dev/kerninst*, a run-time allocated patch area heap, and a runtime-allocated heap of timers and counters (used when the instrumentation code contains performance-gathering annotations). Kerninstd maps both heaps into its address space using mmap on /dev/kmem. To minimize our tool's presence in the kernel, most functionality is in kerninstd. When it needs to perform actions in the kernel's address space, kerninstd enlists the assistance of /dev/kerninst.

This section discusses how KernInst performs fine-grained dynamic instrumentation of a commodity operating system kernel. We present specific examples from our current Solaris implementation; however, we believe that dynamic kernel instrumentation is possible on most modern operating systems.

3.1 Bootstrapping KernInst onto the Kernel

To instrument a running kernel, kerninstd needs to allocate the patch area heap, parse the kernel's runtime symbol table, and obtain permission to write to any portion of the kernel's address space.

Code patches, which hold the dynamically generated code being inserted, are allocated from the patch area heap in the kernel's address space. Kerninstd cannot allocate kernel memory, so it has /dev/kerninst perform the necessary kmem_alloc via an ioctl.

To instrument, kerninstd needs to know where functions reside in memory. Thus, it needs access to the kernel's runtime symbol table. The symbol table on disk (/kernel/unix) is insufficient because it is incomplete; most of the kernel is contained in runtime-loaded modules. /dev/kerninst reads the kernel's runtime symbol table directly from kernel memory on behalf of kerninstd. Solaris provides a similar interface through a driver /dev/ksyms, but it does not label all functions with their associated kernel module.

Both emitting code into the patch area and splicing require write permission to the kernel's address space. Kerninstd writes to the patch area heap directly since it is mapped into its address space (mmap of /dev/kmem). Splicing into existing kernel code is more difficult, because Solaris (like most operating systems) does not allow certain parts of the kernel code to be modified, even from within the kernel. Specifically, the first three Solaris kernel modules, collectively termed the kernel nucleus[2], cannot be written on UltraSPARC platforms because they are mapped only into the I-TLB, and with read-only permission. To write to code in the nucleus, /dev/kerninst maps the appropriate page (using

2. In Solaris 2.x, the first three kernel modules are: *unix*, the architecture-specific part of the kernel; *krtld*, the kernel's runtime linker; and *genunix*, the architecture-independent part of the kernel.

segkmem_mapin), performs the write, and then unmaps it (using segkmem_mapout).

3.2 Structural Analysis

Dynamic code generators perform many of the machine code transformations a compiler does, only at runtime. As such, they would benefit greatly from information that compilers and linkers unfortunately discard in whole or part, such as symbol tables, control flow graphs, and live register analysis. Kerninstd constructs similar information by analyzing the kernel's in-core machine code, creating an interprocedural control-flow graph of basic blocks, and finding live registers at each basic block.

Kerninstd builds a control flow graph of the kernel's machine code by partitioning its functions into basic blocks. This graph is needed for performing live register analysis during dynamic code generation and during splicing. No source code or debugging information is used in this process. First, the runtime symbol table is parsed to determine the in-memory start of all kernel functions. Each function's machine code is then read from memory and parsed into basic blocks. A basic block ends when an instruction that potentially interrupts the program's control flow (except function call) is encountered. Jump tables and jumps to constant addresses are determined from a backwards slice on the register(s) used in the jump. Other register-relative jumps are marked as unanalyzable. KernInst's control-flow graph construction is similar to that done by binary rewriters for user programs, such as EEL [12] and ATOM [20]. However, since KernInst performs its processing at runtime, all code is available, including runtime-loaded modules. Furthermore, since our control-flow graph is interprocedural, more aggressive data-flow analyses are possible.

Next, kerninstd performs an interprocedural live register analysis of the kernel code. For each basic block, the set of live registers at its entry is calculated and stored for later use in code generation and splicing. To conserve storage, kerninstd does not store the live registers for every kernel machine code instruction; such fine-grained analysis is performed during instrumentation as needed.

Figure 2 summarizes the code components of Solaris 2.5.1 running on an UltraSparc. KernInst performs its one-time structural analysis efficiently, as shown in Figure 3. Structural analysis could be optimized by making the results persistent; they only need to be recalculated when a kernel module is loaded or unloaded. However, since the start-up processing time is only about 15 seconds, we have not pursued this optimization.

Kernel Component	Number
Modules	77
Functions	8,457
Basic blocks	107,976
Instruction bytes	2.59 MB

Figure 2: Solaris 2.5 Kernel Overview

Structural Analysis Step	Time
Get kernel runtime symbol table from /dev/kerninst	0.5 sec
Parse functions into basic blocks (create CFG)	6.5 sec
Perform live register analysis on each basic block	8.5 sec
Total	15.5 sec

Figure 3: Structural Analysis One-Time Start-up Costs

3.3 Code Generation Issues

The code generation and splicing phases of dynamic instrumentation are decoupled; kerninstd can splice in code generated from any runtime code generator that can coordinate with kerninstd to overwrite only registers which kerninstd says are free at the instrumentation point, and emit machine code directly to memory at a location specified by kerninstd. The VCODE code generator [7] fits this model well, but many interpreters and runtime compilers do not. Statically generated position-independent code can also be used for instrumentation; at runtime, kerninstd resolves procedure calls, and brackets the code with register spills to ensure that no free registers are overwritten.

Unlike a compiler, KernInst is concerned with inserting (splicing) generated code in the midst of existing kernel code. For safety, dynamically generated code must only write to registers that are free (i.e., do not contain live information) at the instrumentation point. If more registers are required than are free, kerninstd brackets the code with stack frame set-up and tear-down to free up additional registers. On the SPARC, this involves emitting save and restore instructions. Because these instructions cannot safely be executed within the trap handlers for register window overflow and underflow, kerninstd cannot instrument these routines[3].

Figure 4 lists kerninstd's dynamic instrumentation steps. This section describes the first three: live register analysis, allocating a patch to hold the generated code, and code generation.

The first step, live register analysis, determines registers that are available for scratch use at the instrumentation point. Finding live registers is a classic backwards data-flow problem operating on a control-flow graph. Since the set of live registers at the top of each basic

3. We could instrument those routines by explicitly saving live registers to the stack without making use of the SPARC's register window *save* and *restore* instructions. We plan to add this feature in a future version of kerninstd.

Instrumentation Step	Cost	When	Described
1. Finding free registers before and after instrumentation point		The first time code is spliced at this instrumentation point; results are cached thereafter	Section 3.3
Retrieve live registers at the bottom of the basic block (calculated at startup)	66 µs		
Calculate live registers before and after the instrumentation point machine code instruction. Cost is 19 µs per machine code instruction following the instrumentation point in the basic block. (Cost assumes 5 instructions.)	95 µs		
Return result	60 µs		
2. Calculate size of patch & allocate	30 µs	Each instrumentation request	
3. Generate and emit code into patch (add 1 to counter in this example)	79 µs		
4. Emit relocated instruction and (if necessary) jump to instruction following the instrumentation point	27 µs		Section 3.4.1
5. Creating & installing splice to patch (assuming a springboard is required). Unlike the patch area, springboards are *not* mapped into kerninstd for quick writing.			
Allocate springboard	13 µs		Section 3.4.3
Generate springboard code	26 µs		
Write springboard contents to kernel (if springboard is nucleus)	135 µs		
Write springboard contents to kernel (if springboard is not nucleus)	40 µs		
Overwrite 1 instruction at the instrumentation point (if in nucleus)	74 µs		Section 3.4.2
Overwrite 1 instruction at the instrumentation point (if not in nucleus)	35 µs		
Total (worst case: both instrumentation point and springboard in nucleus)	600 µs		
Total (both instrumentation point and springboard not in nucleus)	471 µs		
Total (best case: no springboard needed, instrumentation point not in nucleus)	392 µs		

Figure 4: Dynamic Kernel Instrumentation Main Steps.
Timing measurements taken on a 167MHz UltraSPARC 1 running Solaris 2.5.1.

block was calculated and stored during kerninstd's structural analysis start-up phase, finding live registers at a given instruction within the basic block can be done quickly. The free registers at that point are those that are not live. Live register analysis averages 221 µs in our current implementation.

The second step in code generation allocates patch space that will hold the dynamically generated code. The patch size is the sum of: the size of the machine code being inserted, extra instructions to spill some registers to the stack (if more scratch registers are needed than are available), space for the original instruction at the instrumentation point, and space for a jump back to the instruction following the instrumentation point. If several pieces of instrumentation are inserted at the same instrumentation point, kerninstd simply compounds them in the same code patch. As a rule, there is one code patch for each spliced instrumentation point.

There are usually several possible code sequences (with varying number of instructions) for the returning jump, depending on the required displacement. Thus, the number of instruction bytes required for the code patch cannot be determined until it has been allocated. The circular dependency is broken by assuming the maximum number of instructions needed to perform a jump (4 instructions on SPARC, assuming 32-bit addresses). Space for the code patch is then allocated from the patch heap. Calculating the patch size and allocating it typically takes 30 µs.

Once patch space is allocated, dynamically generated code is emitted. This step could not take place before patch allocation because the machine code representation of PC-relative instructions (such as SPARC's call instruction) depend on their instruction addresses. Because the entire patch area heap was mapped into kerninstd space for writing, generated code is written directly into the patch. For a simple annotation—incrementing a 64-bit integer counter—code generation takes 79 µs. Most of this cost is due to the kernel's policy of deferring page mapping until the first time it is written. Thus, although no explicit kernel calls were made to write to the mapped kernel memory, the kernel was performing noticeable work during the write. Subsequent "warm" writes do not require mapping, and complete in just 7µs. However, since patches are typically written only once, times will normally fall under the slower "cold" value.

3.4 Code Splicing

Fast fine-grained code splicing is KernInst's major technology contribution. Splicing is the action of inserting runtime generated code before a desired kernel code location (the instrumentation point). KernInst's splicing is fine-grained; instrumentation points can be almost any kernel machine code instruction.

Kerninstd splices by overwriting the instrumentation point instruction with a branch to patch code. The code patch contains the dynamically generated code, the

original overwritten instruction, and a jump back to the instruction stream after the instrumentation point. Figure 5 illustrates this basic design.

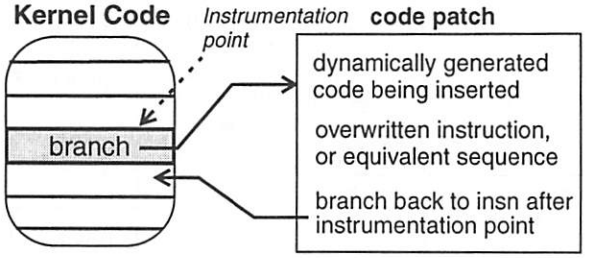


Figure 5: Code Splicing
One machine code instruction is overwritten with a branch to patch code, which contains the desired instrumentation code, the overwritten instruction, and a branch back to the instruction stream.

Ideally, a runtime code splicer should: be fine-grained, able to splice code at any machine code instruction; splice quickly, without the need to pause or synchronize with executing threads; splice without the need for customized code at the instrumentation point (i.e., unmodified kernels); work on multi-threaded kernels; and allow splicing to safely occur at any time, even with threads potentially executing at or near the instrumentation point during splicing. Kerninstd fulfills all of these goals.

Splicing a multi-threaded kernel without pausing requires replacing *only one* machine code instruction at the instrumentation point (with a branch to the code patch). Section 3.4.2 discusses the safety motivation behind single-instruction splicing. Displacement is an issue with single-instruction splicing; branch instructions often have insufficient range to reach a code patch from an instrumentation point. Section 3.4.3 discusses *springboards*, our solution to this problem. But first, we discuss the contents of a code patch.

3.4.1 Code Patch Contents

Following the dynamically generated code being inserted, a code patch ends with the original instrumentation point instruction and a jump back to the instruction after the instrumentation point. Because the original instruction at the instrumentation point is overwritten, it needs to be relocated to the code patch. The relocated instruction is placed after the generated code, so instrumentation code is effectively inserted *before* that machine code instruction. Note that instructions whose semantics are PC-dependent, such as branches, cannot be relocated verbatim to the code patch. In these cases, kerninstd emits a sequence of instructions with combined semantics equivalent to the original instruction.

Patch code ends with a jump back to the instruction following the instrumentation point. If the instrumentation point instruction is an unconditional branch or jump, this step is skipped. If a single branch instruction does not have sufficient range, a scratch register is written with the destination address and then used for the jump. Since this jump executes after the relocated instruction, an available scratch register must be found from the set of registers free after the instrumentation point. This contrasts with instrumentation code, which executes in a context of free registers before the instrumentation point. If no integer registers are available, Kerninstd makes one available by spilling it to the stack[4]. Kerninstd generates the relocated instrumentation point instruction and the returning jump in 36 µs.

Splicing at control transfer instructions having a delay slot requires an extra step. Both the control transfer instruction and its successor (the delay slot instruction) are copied to the code patch. This ensures that the delay slot instruction is executed with the proper semantics (i.e., before the control transfer instruction has changed the PC). An example is shown in Figure 6.

Code Before Splicing

```
tcp_err_ack:
...
0x60060518 call 0x6015b818
0x6006051c mov %i3, %o2
0x60060520 ...
```

Code After Splicing

```
tcp_err_ack:
...
0x60060518 ba,a <patch addr>
0x6006051c mov %i3, %o2
0x60060520 ...
```

Code Patch

```
...dynamically generated code omitted...
call 0x6015b818 // relocated overwritten instruction
mov %i3, %o2 // relocated delay slot instruction
jump to 0x60060520 // return to instruc. after the delay slot
```

Figure 6: Splicing Delayed Control Transfer Instructions.
Both the overwritten instruction (call) and its delay slot instruction (mov) are relocated to the patch. The delay slot instruction left behind will no longer be executed.

Note that when the code patch completes, it returns to the instruction after the delay slot, to ensure that it is not executed twice. As before, if the control transfer instruction was unconditional, there is no need to emit a jump back to the instruction stream, because it would never be executed.

Splicing at the delay slot of a control transfer instruction is difficult because the branch to the code patch will occur before the control transfer instruction

4. A cheaper alternative to spilling an integer register is to store it in an available floating point register. Unfortunately, the SPARC architecture has no instructions for a raw (non-converting) integer-to-floating point register move.

has changed the PC. When the code patch completes, it cannot jump to the instruction following the delay slot; the effects of the control transfer instruction still need to be executed. Unfortunately, there can be two valid return locations if the control transfer instruction is a conditional branch (taken and fall-through). The solution is to effectively relocate the control transfer instruction to the end of the code patch. If this instruction falls through, the code patch returns to the instruction following the delay slot (as usual). This approach works if the instrumentation point instruction is always executed as the delay slot of the preceding control transfer instruction. However, on rare occasions (nine in the Solaris kernel), a delay slot instruction is the target of a branch, and thus is not always executed as the delay slot of the preceding control transfer instruction. Kerninstd does not instrument these cases since a code patch would have to choose from two different instruction sequences for returning. This case is detected by noticing a delay slot instruction at the start of a basic block.

3.4.2 Overwriting a Single Instruction at the Instrumentation Point: Why and How

For safety, kerninstd always splices by overwriting a single instruction at the instrumentation point with a branch to the code patch. After the code patch is written to kernel memory, the instruction at the instrumentation point is overwritten with a branch. It will take time for the new instruction to make its way to the instruction cache; until it does and is fetched from the icache, threads will continue to (safely) execute the original code sequence. Since either the pre-instrumentation or post-instrumentation code sequence is executed (never a mix of both), single-instruction splicing is hazard-free.

Fine-grained splicing by replacing more than a single instruction is inherently unsafe on an unmodified commodity kernel, because a thread can execute a mix of the pre-slice and post-splice sequences. Figure 7a shows the first three instructions of the kernel routine kmem_alloc. The PC of a kernel thread is located before the third instruction, when a two-instruction splice replaces its second and third instructions. (The instrumentation point is the second instruction of kmem_alloc; since two instructions are used in the splice, the third instruction of the function is also replaced.) Since the thread has already executed the sub instruction but not its successor (sra) when the splice occurs, an unsafe sequence of instructions will be executed, as shown in Figure 7b. Note that this problem occurs even in an architecture that can write the two instructions atomically.

In theory, this hazard could be avoided by pausing and performing a backtrace on all kernel threads to check whether execution is currently within, or will return to, one of the instructions being replaced. If a hazard is detected, then splicing is deferred. This strategy is currently used by the Paradyn instrumentation system for user programs [9, 10]. However, it does not work in a kernel, for several reasons. First, pausing the kernel it is not allowed; even if it were, this would involve freezing all of the kernel threads (except, presumably, for the kerninstd thread, which is performing the splice), possibly disrupting critical background activities. Second, performing the necessary backtrace on all threads (of which there are thousands in the Solaris kernel) would be expensive. Third, even if pausing were possible and practical, a jump with an unanalyzable destination (such as a longjmp) may jump to the middle of a splice sequence, resulting in the execution of an inconsistent code sequence. A fourth problem with multiple-instruction splicing occurs when the instrumentation point is at the end of a basic block B_1. Some of the splice sequence spills over into the next basic block B_2. If B_2 is reachable from another block (say, by a branch from block B_3), then code taking the path (B_3, B_2) will execute the second half of a splice sequence, an inconsistent code sequence which can crash the kernel. For these reasons, we conclude that fine-grained dynamic instrumentation is unsafe when using multiple-instruction splicing.

Single-instruction splicing is difficult on architectures whose branch instructions always have a delay slot, such as MIPS. This forces us to use a delayed branch instruction for splicing, resulting in an unusual execution sequence when jumping to patch code. The instruction following the instrumentation point implicitly becomes the delay slot of the splice branch instruction, and thus is executed before the code patch is reached. In particular, the instruction after the instrumentation point is executed before the instruction origi-

Code Before Splicing

```
kmem_alloc:
0x1006e768 save %sp, -0x60, %sp
0x1006e76c sub %i0, 0x1, %l2
0x1006e770 sra %l2, 3, %i5      <-- PC
```

Code After Splicing

```
kmem_alloc:
0x1006e768 save %sp, -0x60, %sp
0x1006e76c call <code patch>
0x1006e770 nop                  <-- PC
```

(a) Dual-Instruction Splicing: Before and After

Executed Sequence

```
save %sp, -0x60, %sp
sub %i0, 0x1, %l2
nop
```

(b) Inconsistent Instruction Sequence

Figure 7: Why Multiple-Instruction Splicing is Hazardous

nally at the instrumentation point, which gets relocated to the code patch. Thus, the execution ordering of the instruction originally at the instrumentation point and its successor are reversed. In some cases, this can be worked around. If the instrumentation point instruction and its successor are mutually independent, then reversing their execution order is safe. If not, but the instrumentation point instruction is independent of its successor, and if the successor instruction is idempotent, then both the instrumentation point instruction and its successor can be placed in the code patch. The resulting execution sequence is (1) successor, (2) original instrumentation point instruction, (3) successor. Under the mentioned constraints, step (1) is equivalent to a nop. Of course, the independence and idempotency constraints will not always be met, making single-instruction splicing on always-delayed-branch architectures difficult. A final possibility is to splice by replacing the instrumentation point instruction with a trap or illegal instruction. This will immediately jump to a trap handler, which, if it can be safely instrumented, can check the offending instruction's PC and perform a long jump to the appropriate code patch. Note that the current implementation of kerninstd is on the SPARC v9 architecture, which has a non-delayed branch instruction (ba,a) that we always use for splicing, thus avoiding these difficulties.

Single-instruction splicing on variable-instruction-length architectures such as x86 is challenging; depending on the existing code at the instrumentation point, a jump instruction used in splicing (5 bytes on x86) may not overwrite exactly one instruction. If the jump instruction is smaller than the instruction being overwritten, then the new instruction stream will contain the newly written jump instruction, followed by the tail end of the original instruction (which will never be executed). On the other hand, when the instrumentation point contains an instruction that is smaller than the jump instruction, splicing overwrites more than one instruction. This can cause problems because the next instruction will also be overwritten; if that instruction is the destination of a branch, a corrupted instruction stream can be executed. This case can be handled by splicing using a one-byte trap or other illegal instruction. This will transfer control to a trap handler, which, if it can be instrumented using the more conventional jump instruction, can be made to look up the address of the offending instruction in a hash table, undo any processor-state side effects of the trap, and transfer control to the appropriate code patch. Since there are single-byte trap instructions, it is always possible to overwrite just one instruction, and is thus hazard-free.

Our current implementation of KernInst on the UltraSPARC splices instrumentation points outside the kernel nucleus in 35 µs, via a pwrite to /dev/kmem. If the instrumentation point is within the nucleus, kerninstd has /dev/kerninst perform the necessary map, write, and unmap sequence (see Section 3.1), which completes in 74 µs.

3.4.3 Springboards: Why and How

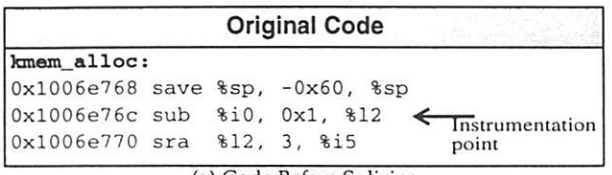
We have seen that safety requires single-instruction splicing, but RISC architectures do not provide an ideal instruction to branch from any instrumentation point to a code patch. An ideal splicing instruction is one which: has enough displacement to reach the patch from the instrumentation point; has no delay slot, which would cause a second instruction to be executed before the code patch is reached; has no side effects other than changing the PC; and is absolute or PC-relative, but not register relative. (Although register relative jumps can reach any part of an address space, they require a register to be set before use. This leads to hazardous multiple-instruction splicing.) Figure 8 reviews the features of branch and jump instructions for several architectures that are best suited for single-instruction splicing.

Arch	Instruction	Range	Delay Slot?	Side Effects
SPARC v9	call	PC ± 2 GB	yes	writes *o7*
	ba,a	PC ± 8 MB	no	none
	jump	register ± 16K	yes	none
PowerPC	b	PC ± 32 MB	no	none
MIPS IV	j	current 256MB aligned region	yes	none
	b<cond>	PC ± 128K	yes	none
Alpha	branch	PC ± 4MB	no	none
	jmp	register ± 16K	no	none
x86	jmp	PC ± 2 GB	no	none

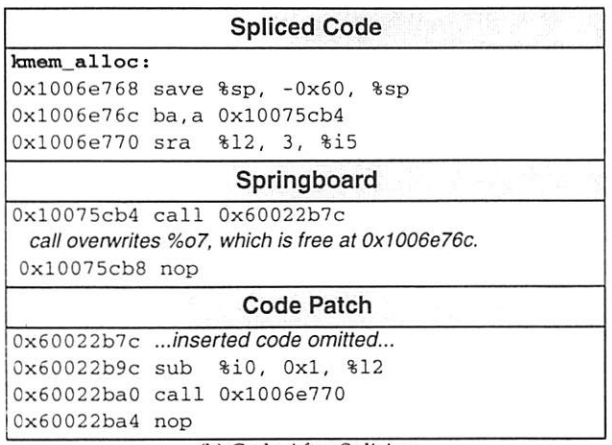
Figure 8: Suitability of Various Instructions for Single-Instruction Splicing.
None of the RISC architectures has an ideal splicing instruction

Unfortunately, none of the RISC architectures has an instruction that is always suitable. The key limitation is displacement. The patch area heap is allocated arbitrarily far from the code of most kernel modules. Thus, we need a means for reaching a patch, no matter the required displacement, while still splicing with a single instruction (for safety).

KernInst implements a general solution to the displacement problem called *springboards*. A springboard is a scratch area that is reachable from the instrumentation point by a suitable jump instruction. The idea is for the splice instruction to merely branch to an available nearby springboard, which in turn takes as many instructions as needed to jump to the code patch. Figure 9 shows an example of code splicing in the presence of springboards. Like the code patch, the springboard is written before the branch instruction is written



(a) Code Before Splicing



(b) Code After Splicing

Figure 9: Code Splicing Using a Springboard
The instrumentation point (at 0x1006e76c) is too far from the code patch (at 0x60022b7c) to be reached with a branch, so kerninstd instead places a branch to a nearby springboard (at 0x10075cb4), which in turn performs a multi-instruction long jump to the code patch.

at the instrumentation point, so no kernel thread will execute springboard code until the splice has completed. Thus, the safety properties of single-instruction splicing are maintained.

The springboard approach requires chunks of scratch space (collectively, the *springboard heap*) to be conveniently located at various spots in the kernel, so that every kernel instruction can reach the nearest chunk when using one of the suitable jumps of Figure 8. Fortunately, UNIX SVR4.2-based kernels (including Solaris), Linux, and Windows NT all have ideally suited available scratch space: the initialization and termination routines for dynamically loaded kernel modules.

In a kernel that allows modules to be loaded at runtime (and unloaded, when memory is tight), each module has initialization and termination routines that are called just after the module is loaded, and just before it is unloaded, respectively[5]. Kerninstd locks kernel modules in memory, which guarantees that the initialization and termination routines will no longer be called; this makes them free to use as springboard space. In addition, preventing module unloading and reloading obviates the need to re-insert splicing code changes that would be lost when and if a module gets re-loaded. In practice, no single kernel module approaches one megabyte in size, so even a jump instruction with a modest range, such as SPARC's ba,a, can easily reach the nearest springboard.

In Solaris, the first three kernel modules (the nucleus) are not subject to runtime loading or unloading, and thus do not have initialization and termination routines. Furthermore, in practice, the nucleus modules are loaded into kernel virtual memory far from the dynamically loaded ones, and thus cannot reach the latter's initialization and termination routines as potential springboard space. However, two routines within the nucleus, _start and main, are invoked only while the kernel is booting. Since they are never again executed, kerninstd adds these routines to the springboard heap. Figure 10 summarizes the springboard space set aside by our current Solaris 2.5.1/UltraSPARC implementation of kerninstd.

Location	Size (bytes)
Nucleus (_start and main)	884
Outside nucleus (initialization & termination routines of kernel modules)	7128

Figure 10: Available Springboard Space in Solaris 2.5.1

While the springboard technique may seem ad-hoc, it is applicable to most kernels with which we have experience. Furthermore, with 64-bit operating systems running on an architecture with 32-bit instructions, there is no possibility of finding a single branch instruction with 64 bits of displacement. Because they are not limited to a single instruction, springboard code can have arbitrary displacement, making them a general solution to the problem of reaching patches.

If a springboard is needed, kerninstd allocates it and generates its contents in 39 µs. Its contents are then copied to kernel space. If the springboard does not reside within the kernel nucleus, kerninstd fills it with a single pwrite to /dev/kmem in 40 µs. If the springboard is within the nucleus, kerninstd invokes /dev/kerninst to perform the write (see Section 3.1) in 135 µs.

4 Using Dynamic Instrumentation for Performance Profiling

This section presents a case study using KernInst to locate performance bottlenecks in the kernel and in an application, using a Web proxy server as the workload.

5. In Solaris 2.x, these routines are called *_init* and *_fini* in each module. In the SVR4.2 UNIX standard, these routines are called *<module>_load* and *<module>_unload*. In Linux, they are called *init_<module>* and *cleanup_<module>*. In Windows NT, device drivers have a *DriverEntry* routine which also installs a pointer to a cleanup routine.

4.1 Kernel Metrics

We used KernInst to implement a kernel profiling tool. Due to space constraints, we discuss only a representative sample of its performance primitives. These include basic counters, cycle timers, accumulators, and average-over-time (AOT) accumulators. The primitives are non-blocking (and thus safe in a multi-threaded environment) by using the compare-and-swap instruction when changing their values. These primitives can be combined and used with more complex control flow code. Basic counters are implemented by inserting increment operations at the appropriate point. Cycle timers insert start and stop operations in the code, often at function or basic block entries and exits. Accumulators collect values from kernel variables or hardware counters (such as icache misses) to calculate a total. AOT accumulators calculate the average value of some counter or variable over time, such as the average number of threads executing in a given function, or the average number of threads waiting for a condition variable. They use instrumentation code to calculate the area under the curve of the value being averaged. The area is calculated by instrumentation code that sums rectangles each time the event value changes.

Performance *metrics* are formed by applying the primitives. Four metrics in our tool-kit library are call counts, average number of executing threads, average number of waiting threads, and virtual timers. Call counts simply record the number of times a function is called. The "calls made to" curve in Figure 11 shows an example of the number of calls to the kernel function copen. Average number of executing threads uses a counter to record the number of threads in a section of code (incremented at entry to the block and decremented at exit) and a value recording the last time the value being averaged has changed; the AOT is calculated using these values. The "concurrency" curve in Figure 11 shows an example of the average number of threads in function copen. Average waiting threads counts the number of threads waiting on a kernel mutex variable (kmutex_t), by instrumenting mutex_enter; the AOT primitive is applied to this counter. The underlying processor cycle counters used by KernInst measure elapsed (wall) time. Virtual (processor) time can be important in trying to isolate bottlenecks. KernInst measures virtual time by instrumenting the context switch handler; we can detect when a kernel thread is dispatched, and stop and re-start the timer primitive at the appropriate times.

Metrics can be constrained to a given process or kernel thread by predicating the primitives. To find the average number of kernel threads belonging to process P executing within function F over time, the AOT primitives spliced into the entry and exit points of F are predicated with "if current pid equals P then...".

SPARC compilers make instrumenting before a function's return challenging. If a function B (called from A) ends by calling C, then B is (tail call) optimized to unwind its stack frame in the delay slot of the call to C; this makes C return directly to A. To instrument at the exit point of B, KernInst first splices in a code sequence that de-optimizes the tail call sequence. This was first done by Paradyn [10].

4.2 Web Proxy Server Benchmark

We used KernInst to study the performance of Solaris running version 1.1.22 of Squid, a Web proxy server. We used version 1.0 of the Wisconsin Proxy Benchmark [1] running 30 client processes to drive Squid. All Squid files were stored on the local disk running the Unix File System (UFS). KernInst and Squid were run on the same machine, an UltraSPARC 1 with 128 MB of RAM running Solaris 2.5.1. Because KernInst instruments entirely at runtime, gathering kernel performance information was an interactive process.

Previous studies of proxy servers have shown file opens to be a common bottleneck, so we first measured

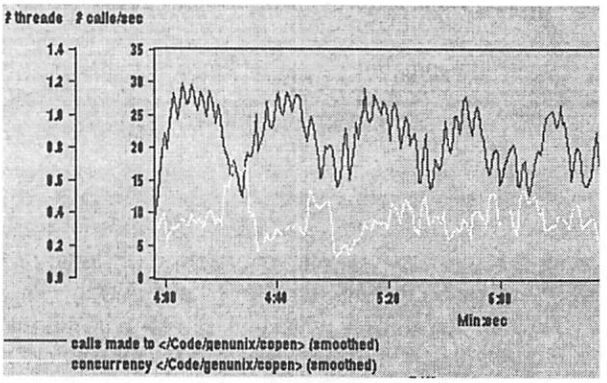


Figure 11: Squid file creation
Although called only 20-25 times/second, copen is a clear bottleneck

the kernel function copen, which handles both file open and file creation. As shown in Figure 11, it is called 20-25 times per second, and averages about 0.4 threads executing within this code at any given time. Since Squid is a single-threaded program, this means that on average, 40% of Squid's elapsed time was spent either opening existing files for reading, or creating new files for writing.

copen performs two major calls: falloc to allocate an entry in the process's file descriptor table, then vn_open for file system-specific opening. Because Squid maintains one disk file per cached HTTP object, we expected falloc to be a bottleneck because it performs a linear search to find an available table entry. However, we

found falloc to consume negligible run time; most of copen's time was spent in vn_open.

vn_open has two paths, one for creating files (where it calls vn_create), and one for opening files. We found that the vn_create path, although called only 8 times per second, accounted for almost all of vn_open's bottleneck. Thus, file creation (which, in Squid's case, is a call to open() with the O_CREAT flag) is the bottleneck.

vn_create calls two important routines: lookuppn and ufs_create. lookuppn translates a full path name into a vnode structure. ufs_create creates a UFS file when given a vnode. Measurements for these routines are shown in Figure 12; it shows that *both* are bottlenecks.

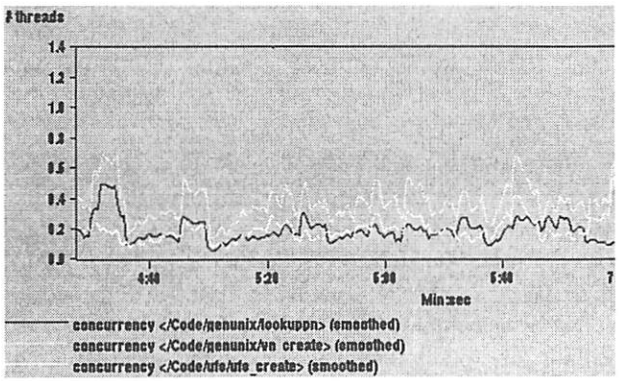


Figure 12: vn_create and its components
Both lookuppn and ufs_create are (distinct) bottlenecks

lookuppn, better known as namei, obtains a vnode by parsing path components one by one and calling the file system's lookup routine (ufs_lookup for UFS) for each. In the general case, ufs_lookup must go to disk to obtain an inode location from a directory file, and to read the inode contents. To optimize path name lookup, Solaris uses a *directory name lookup cache*, or DNLC, which hashes path name components to entries in an *inode cache* [5]. A DNLC hit bypasses both reading the directory file (ufs_dirlook) and reading the inode (ufs_iget). By dynamically instrumenting the kernel to count calls to ufs_dirlook and to lookuppn, we found that the DNLC hit rate was about 90%. Nevertheless, the miss penalty (execution of ufs_dirlook) was high enough to account for the ufs_lookup bottleneck, as shown in Figure 13.

Squid's preponderance of small cache files (over 6,000 in our benchmark) overwhelmed the DNLC, which contains 2,181 entries by default. To address the bottleneck, we increased the DNLC size to the maximum allowed value of 17,498. As shown in Figure 14, this eliminates the ufs_lookup bottleneck; it now accounts for just 1% of Squid's elapsed time. Two permanent solutions suggest themselves. First, the DNLC should grow when needed, to avoid conflict misses. A

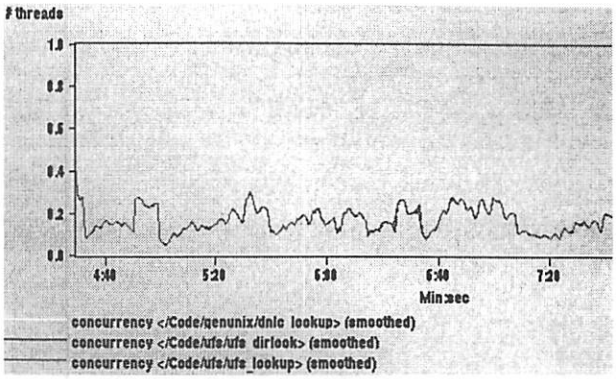


Figure 13: ufs_lookup
ufs_dirlook, called on a DNLC miss, accounts for the ufs_lookup bottleneck (the curves almost entirely overlap). Note that dnlc_lookup time is essentially zero.

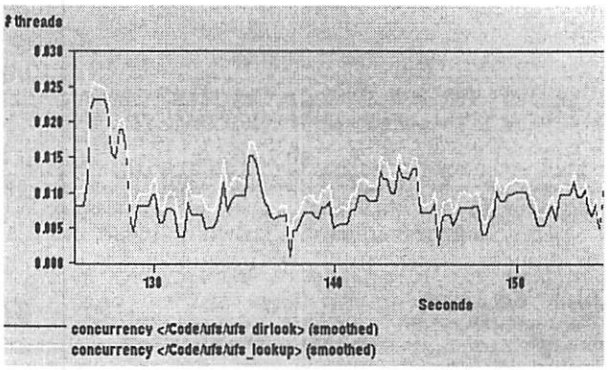


Figure 14: The effect of increasing DNLC size
The miss penalty routine (ufs_dirlook) is now called so infrequently that it now accounts for just 1% of Squid's elapsed time.

flip side argument is that Squid simply uses too many small files, and should be redesigned to use one large fixed-size file for its disk cache.

Recall from Figure 12 that ufs_create was the second bottleneck in vn_create, accounting for 20% of Squid's elapsed time. Almost all of ufs_create's time is spent in ufs_itrunc, which is invoked because Squid passes the O_TRUNC flag to the open system call. Thus, about 20% of Squid's time is spent truncating existing cache files to zero size when opening them. Most of ufs_itrunc's time is spent in ufs_iupdat, which synchronizes updates to inodes. Thus, truncation is slow because UFS synchronizes changes to meta-data. Squid reuses obsolete disk cache files rather than deleting the obsolete file and creating the new one from scratch. The motivation is to avoid expensive meta-data operations required in file deletion (updating the parent directory file and freeing inodes). But the ufs_itrunc bottleneck shows that Squid's strategy is backfiring.

To address this bottleneck, we note that after deleting all of the file's inodes in truncation, some will be added (again, synchronously) as the new version of the

file is written. If the new file size is at least equal to the old size, then the inode deletions and creations amount to a very expensive no-op. If the new file size is less than the original size, a (lesser) optimization is to only delete the inodes at the file's end that are no longer needed. We modified Squid to implement these changes; their effect is shown in Figure 15. The time spent synchronously

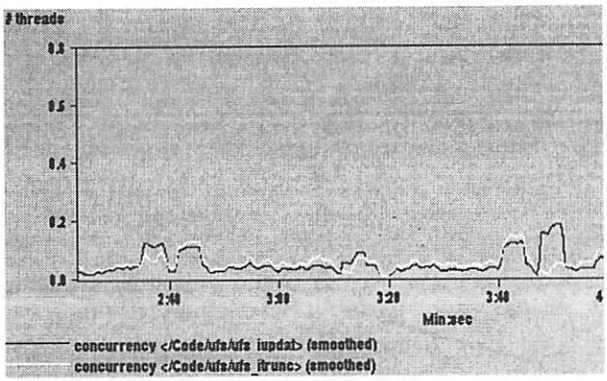


Figure 15: The effect of improving Squid's truncation
ufs_itrunc now accounts for less than 10% of Squid's elapsed time

updating inodes has been reduced from 20% of Squid's run time to about 6%.

The combined effect of the two optimizations are shown in Figure 16. The open system call, which once

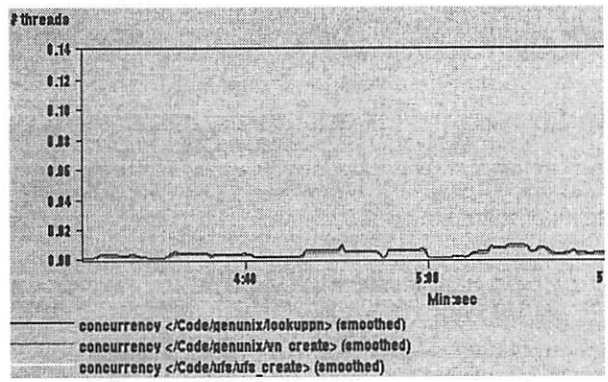


Figure 16: vn_create performance with both optimizations
The lookuppn component has been reduced via a larger DNLC; the ufs_create component has been reduced because the open system call no longer uses the O_TRUNC flag. Note that ufs_create time is essentially zero.

consumed 40% of Squid's run time, now takes only 1%. To this must be added the time spent truncating cache files (which is now done explicitly instead of through the O_TRUNC flag to open); from Figure 15, this is about 6%. Thus, what once took 40% of Squid's elapsed time now takes 7%.

5 Safety and Security Issues

Splicing code into a running kernel can introduce race conditions when kernel threads are executing at an instrumentation point being spliced (splicing hazards), adversely disrupt kernel execution (safety violations), or introduce undesirable information flows (security violations). Single-instruction splicing (Section 3.4.2) solves the splicing hazard problem. We plan to harness the rich body of existing work in safety and security, but in a commodity kernel (that was *not* written with extensibility in mind) with fine-grained patching, these issues become more complex. Below we summarize the issues and outline our initial directions.

Trusted code. In this approach, an authority certifies code as well-behaved. The certification may be that the generator of the code or the code itself lies in a specially protected directory that only a system administrator can update. Alternatively, the code may come from a trusted party on the network with a digital signature. This strategy is the fall-back position in our current work; kerninstd currently requires that applications written on top of it have super-user privileges.

Dynamically safe code. Two safety issues must be addressed when executing inserted code. First, safety violations must be detected, by inserting extra instructions into the dynamic code. Second, corrective action must be taken after a violation is detected. This action might be as simple as exiting from the dynamic code or killing the kernel thread that executed the offending code, or it might be much more complex. If the dynamic code is *local*, executing on behalf of a single user or process and not modifying any shared kernel data structures, then exiting the dynamic code is sufficient. Software fault isolation (SFI) techniques [21] augment the dynamic code with run-time checks to insure the code stays within a local space. VINO [17] and Exokernel [6] are two systems that use this technique. The dynamic code might be *global*, in common parts of the kernel and accessing data structures shared by other kernel threads. In this case, terminating the offending code may leave locks held or shared data structures in an inconsistent state. SFI techniques would need to be significantly extended to handle shared resources. Using a kernel whose data structures have transaction semantics (such as in VINO) might simplify constructing a recovery mechanism.

Statically safe code. Code that can be identified statically as safe has two advantages. First, the code is potentially more efficient since no run-time checks are needed. Second, since the code can never misbehave, no recovery scheme is needed. Proof carrying code (PCC) [14] is an example of this approach. However, PCC requires a safety policy to be formally defined for all resources accessed by the extension. Thus, inserted code can only access kernel data structures and call kernel

functions that have gone through the rigor of a formally defined safety policy.

Combined approaches. A combination of static and dynamic checking, such as done with "safe" languages such as Java and Modula-3 in SPIN [3], potentially requires fewer run-time checks, but still needs a recovery strategy.

Extension code can be classified by its interaction with the underlying kernel. *Annotations*, such as performance measurement and debugging code, are orthogonal to the underlying computation. If an annotation only writes to its own data, does not transfer control outside of the annotations code, and is bounded in its time and resource requirements, a recovery strategy is easy: the annotation can be removed. Annotations that call other kernel functions (such as locking routines) may temporarily modify system state. Safety in such annotations requires a specification of the semantics of the kernel routines that the dynamic code calls [15]. We plan to design specifications that cover common synchronization scenarios. Recovery in a commodity operating system after an open annotation fault is an area we are actively researching.

Code *adaptations* intentionally change the behavior of the underlying system in some way. Examples include on-the-fly optimizations such as specialization [16] and outlining [13]. An adaptation may take some part of the kernel and replace it with code that accomplishes the same task, but in a more efficient or reliable manner. We are currently developing the mechanisms for closed-looped dynamic measurement and optimization. Adaptations may also include adding new functionality into the kernel. The safety and recovery issues for fine-grained adaptations are more complex than for open annotations.

Security issues are distinct from safety issues. Security is restricting information flows and authenticating data modifications. Annotations and adaptations may be efficient and safe without being secure. For example, sensitive kernel structures or process address spaces could be copied into a file quickly and safely, but open large security holes. Security can be addressed much like safety, by verifying that formally-defined policies for all resources accessed by inserted code are respected.

6 Related Work

Extensible operating systems such as SPIN [4], Exokernel [6], and VINO [17] allow processes to download code into a kernel, but differ from our approach in several ways. First, they are not unmodified commodity kernels. Second, they perform coarse-grained instrumentation; for example, VINO, allows C++ classes to customize object methods [19]. Third, the limited number of instrumentation points are pre-coded in a way that allows easy instrumentation; for example, Synthetix [16] replaces a function that is called through a pre-existing level of indirection by overwriting the appropriate function pointer. Requiring special code where process-specific customization *might* take place incurs a small overhead on methods that are not customized. It also limits the granularity of instrumentation because it would be impractical to place the level of indirection in every kernel basic block. We note that KernInst is complementary to, and could be used with these research kernels to provide additional splicing capabilities.

Digital's Continuous Profiling system (dcpi) [2] measures detailed performance metrics (such as cycles and icache misses) at the instruction level of a commodity kernel. Unlike KernInst, dcpi does not instrument kernel code in any way; this precludes metrics that cannot be readily sampled. KernInst could be used in concert with continuous profiling to create additional metrics in software.

Paradyn [10] dynamically instruments user programs. Our work differs from Paradyn in several ways: it applies to kernels; instrumentation is fine-grained, whereas Paradyn limits instrumentation points to function entries, exits, and calls sites; and KernInst instruments without pausing, whereas Paradyn incurs substantial overhead by pausing the application and walking the stack to ensure safe splicing for each instrumentation request.

Static binary rewriters such as EEL [12] and ATOM [20] are fine-grained and allow arbitrary code to be inserted into user programs (and potentially to kernels). Because static rewriting requires the program to be taken off-line during instrumentation, one must instrument everything in case it may turn out to be of interest. By contrast, dynamic instrumentation allows the user to refine, at runtime, what instrumentation is of interest.

Kitrace [11] traces kernel code locations. It replaces instructions being traced with a trap, which transfers control to a custom handler. This handler appends an entry to the trace log and resumes execution. Because trap instructions can be inserted at most kernel instructions, kitrace is fine-grained. Kitrace differs from our work in several ways: it requires a kernel recompile; it does not insert general code into the kernel; and its method of resuming execution after a trap is more expensive than in dynamic instrumentation. Fine-grained dynamic instrumentation subsumes kitrace because it can insert arbitrary code, not just trace-gathering code.

SLIC [8] provides extensibility in commodity operating systems by rerouting events crossing certain kernel

interfaces (system calls, signals, and virtual memory routines) to extensions that have either been downloaded into the kernel, or run in a user-level process. SLIC interposes extensions on kernel interfaces by rewriting jump tables or through binary patching of kernel routines. When it performs binary patching, SLIC replaces several instructions at the start of a kernel function; as we have seen in Section 3.4.2, multiple-instruction patching is unsafe. In addition, SLIC is not fine-grained; for example, interposing system calls provides for only a few dozen kernel instrumentation points.

7 Conclusion and Future Work

Fine-grained dynamic kernel instrumentation has many uses, including performance profiling, debugging, testing, optimizing, and extending the kernel. In this paper, we have shown a design and implementation of dynamic kernel instrumentation, which combines fine-grained splicing with dynamic code generation. We have shown this technology to be feasible by implementing it on Solaris 2.5.1 running on an UltraSPARC, and we are investigating ports to other architecture/OS combinations, including x86.

While KernInst opens up many areas of opportunity, for general use, it creates safety and security concerns. We are currently formulating a formal access and control model, with the goal of automating much of the checking the dynamic code.

Acknowledgments

We thank Pei Cao and Kevin Beach of the WisWeb group for supplying the Wisconsin Proxy Benchmark; Stephen Chessin and Madhusudhan Talluri of Sun Microsystems for technical assistance and suggestions; and Matt Cheyney, Carlos Figueira, Karen Karavanic, Tia Newhall, Brian Wylie, and Zhichen Xu for their comments on this manuscript.

References

[1] J. Almeida and P. Cao. Wisconsin Proxy Benchmark 1.0. `http://www.cs.wisc.edu/~cao/wpb1.0.html`.

[2] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.-T.A. Leung, R.L. Sites, M.T. Vandervoorde, C.A. Waldspurger, and W.E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? *16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, Oct. 1997.

[3] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, Effective Dynamic Compilation. *ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, May 1996.

[4] B.N. Bershad, S.Savage, P. Pardyak, E. Sirer, M. Fiucynski, D. Becker, C. Chambers, and S.N. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. *15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, Dec. 1995.

[5] A. Cockcroft and R. Pettit. *Sun Performance and Tuning: Java and the Internet.* Sun Soft Press, 1998.

[6] D.R. Engler, M.F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. *15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, Dec. 1995.

[7] D.R. Engler. VCODE: a Retargetable, Extensible, Very Fast Dynamic Code Generation System. *SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, May 1996.

[8] D. Ghormley, S. Rodrigues, D. Petrou, and T. Anderson. SLIC: An Extensibility System for Commodity Operating Systems, *1998 USENIX Technical Conference*, New Orleans, June 1998.

[9] J.K. Hollingsworth, B.P. Miller and J. Cargille. Dynamic Program Instrumentation for Scalable Performance Tools, *Scalable High Performance Computing Conference*, Knoxville, May 1994.

[10] J.K. Hollingsworth, B.P. Miller, M.J.R. Gonçalves, O. Naim, Z. Xu and L. Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. *International Conference on Parallel Architectures and Compilation Techniques*, San Francisco, Nov. 1997.

[11] G. H. Kuenning. Precise Interactive Measurement of Operating Systems Kernels, *Software—Practice & Experience* 25, 1 (January 1995).

[12] J.R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. *ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*, La Jolla, CA, June 1995.

[13] D. Mosberger, L.L. Peterson, P.G. Bridges, and S. O'Malley. Analysis of Techniques to Improve Protocol Processing Latency. *ACM SIGCOMM 1996*, Stanford, CA, Aug. 1996.

[14] G.C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Oct. 1996.

[15] G.C. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. ACM *SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998.

[16] C. Pu, T. Audrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. *15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO. Dec., 1995.

[17] M.I. Seltzer, Y. Endo, C. Small, and K.A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Oct. 1996.

[18] M.I. Seltzer and C. Small. Self-monitoring and Self-adapting Operating Systems. *6th Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, May 1997.

[19] C. Small. A Tool for Constructing Safe Extensible C++ Systems. 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), Santa Fe, NM, April 1998.

[20] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. *ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)*, Orlando, FL, June 1994.

[21] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-based Fault Isolation. *14th ACM Symposium on Operating Systems Principles (SOSP)*, Asheville, NC, Dec 1993.

ETI Resource Distributor: Guaranteed Resource Allocation and Scheduling in Multimedia Systems

Miche Baker-Harvey

Equator Technologies, Inc.
520 Pike Street, Suite 900
Seattle WA 98101-4001
miche@equator.com

Abstract

Multimedia processors offer a programmable, cost-effective way to provide multimedia functionality in environments previously serviced by fixed-function hardware and digital signal processors. Achieving acceptable performance requires that the multimedia processor's software emulate hardware devices.

There are stringent requirements on the operating system scheduler of a multimedia processor. First, when a user starts a task believing it to be backed by hardware, the system cannot terminate that task. The task must continue to run as if the hardware were present. Second, optimizing the Quality of Service (QOS) requires that tasks use all available system resources. Third, QOS decisions must be made globally, and in the interests of the user, if system overload occurs. No previously existing scheduler meets all these requirements.

The Equator Technologies, Inc. (ETI) Resource Distributor guarantees scheduling for admitted tasks: the delivery of resources is not interrupted even if the system is overloaded. The Scheduler delivers resources to applications in units known to be useful for achieving a specific level of service quality. This promotes better utilization of system resources and a higher perceived QOS. When QOS degradations are required, the Resource Distributor never makes inadvertent or implicit policy decisions: policy must be explicitly specified by the user.

While providing superior services for periodic real-time applications, the Resource Distributor also guarantees liveness for applications that are not real-time. Support for real-time applications that do not require continuous resource use is integrated: it neither interferes with the scheduling guarantees of other applications nor ties up resources that could be used by other applications.

1 Introduction

We have designed and implemented a multimedia resource manager and scheduler for managing resources on our MAP1000 processor. The processor is designed to execute applications that emulate such fixed-function hardware as MPEG video encoders and decoders, 2D and 3D graphics engines, audio devices, and modems.

The MAP1000 is a multimedia processor comprised of a Very Long Instruction Word (VLIW) processor with a RISC-like instruction set; a multi-element Fixed Function Unit (FFU); and a programmable, multi-ported DMA engine, called the Data Streamer. The VLIW processor can issue up to four operations each cycle. Figure 1 shows a block diagram of the MAP1000.[1]

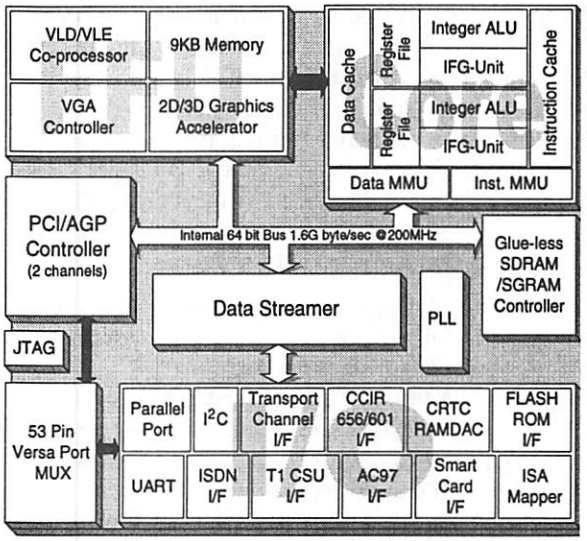


Figure 1: Block Diagram of the Map1000.

The MAP1000 runs the MMLite operating system, part of the Talisman program's Escalante reference design from Microsoft. The operating system supports basic functions with a COM-style interface. As delivered, the operating system supported a subset of the features based on the Rialto scheduler. We have

[1] Several additional features make the MAP1000 extremely effective at running multimedia applications. However, since these features do not raise issues for the Resource Distributor, they are not discussed in this paper. For more information, see [Basoglu et al 99].

removed these features and implemented the ETI Resource Distributor (RD) in their place.[2]

Our goal is to provide an environment in which applications can emulate hardware. We support real-time guarantees that are harder than conventional soft real-time guarantees: user expectations for hardware do not allow for failures nearly as often as would occur with a truly soft real-time scheduler. We support conventional tasks running concurrently.

We provide absolute scheduling guarantees in the face of a dynamic task set. A real-time task admitted to our system is guaranteed to receive predictable resources. This guarantee applies in every period, whether the system is overloaded or not.

Decision making about resource distribution policy is well-integrated. Policy decisions are made globally, not by any single application. The policy delivered is affected neither by accidents of timing nor by the order of task creation. Scheduling guarantees are maintained while policy decisions are being made.

While the RD was conceived in an environment that required harder real-time guarantees, it is appropriate for many systems that combine conventional and soft real-time tasks. Its stricter real-time guarantees, and its integrated support for making global policy decisions, make it attractive for real-time applications. Conventional tasks exist concurrently, and their performance is also constrained by the global policy.

2 Organization of This Paper

Section 3 of this paper describes the design of the ETI Resource Distributor (RD). Section 4 describes implementation algorithms for the primary RD components. Additional features and ancillary issues pertaining to the ETI RD are detailed in Section 5, and Section 6 quantifies performance. Section 7 draws conclusions and presents opportunities for future research and development in the area of multimedia resource managers and schedulers.

In this paper, the term *application* refers to some piece of code started by a *user*, where the *user* is someone sitting at the system. The application may be comprised of one or more *threads* or *tasks,* terms used interchangeably. Both are schedulable entities on the MAP1000.

3 ETI Resource Distributor Design

This section describes various aspects of the ETI RD design. First, we attempt to characterize multimedia applications that run on the MAP1000. We then itemize design requirements abstraced from these applications. Next, we present ETI RD design components and follow them with a discussion of alternative designs. We conclude the section with a synopsis of the benefits offered by our design.

3.1 Application Characteristics

The ETI RD manages resources in a way that maintains for the user the appearance of actual hardware rather than of software running on a separate processor. In particular, applications must degrade gracefully in overload, and one application cannot cause unpredictable behavior in another. MAP1000 applications share some of the following characteristics that helped to drive this design:

1. They are primarily periodic, with naturally or externally set periods.

Some applications are strictly periodic, such as MPEG, whose period is defined by the content provider. Others have a period defined by actions that occur only at specific points; for example, the output for 2D graphics is paced by the screen refresh rate set by the user. While the periods we support may be as small as 500 µSec, the CPU requirements within a period tend to be relatively high. For example, the AC3 audio task requires about 12% of the core VLIW processor cycles.

2. They are capable of shedding load when the system is overloaded.

The applications can shed load when there are insufficient resources to run all tasks at the highest quality. For instance, the MPEG decoder can drop frames or change resolution to use fewer resources.

3. Their resource requirements are discrete, not continuous.

The work that an application must do is predictable. For example, processing an MPEG frame to a given resolution requires a known amount of CPU time. If more is allotted, it will be wasted; if less is allotted, the frame will not be decoded in time. To shed load, the MPEG application drops some complete frames or alters the resolution of the output. It is not useful for MPEG to process part of a frame.

Resource requirements for the MPEG application take quantum steps, from a maximum CPU requirement for top quality (displaying all frames at full resolution) to a lowest CPU requirement for the poorest quality supported (dropping frames and/or reducing resolution). The application can make only step-wise degradations.

This is not the case for all applications. Both 2D and 3D graphics are notable exceptions: the work they must do is a function of the complexity of the scene to be rendered, which is not known far in advance.

4. Their performance is extremely well characterized.

At a low level, the performance characteristics of these applications are well understood. On the MAP1000, the inner loops are understood down to the cycle.

[2] Information on the MMLite operating system can be found in [Jones et al. 96] and [Helander & Forin 98].

The task set is dynamic, with most new applications being started by user request. Generally, these applications can be denied service if the system does not have the resources to run them. Some applications cannot be denied service, or the user does not want them to be denied service. A telephone-answering modem task is an example.

3.2 Design Requirements

The ETI Resource Distributor has three design requirements (or "first principles") derived from the nature of our multimedia applications and user requirements. First, the RD must not cause performance anomalies in tasks that were started by the user. Second, it must allocate (nearly) 100% of available resources to ready tasks. Third, the quality of service (QOS) policy is determined by the user. Each of these requirements is now described in further detail.

1. Once a task has been successfully started by the user, it must continue (from the user's perspective) until it terminates naturally or is terminated by the user.

Assume that the user initiates a task (e.g., hitting the "play" button on the CD player). Once the task has begun (e.g., the sound track begins to emanate from the speakers), it will continue until it terminates (the end of the CD is reached) or until the user terminates it (e.g., by hitting "stop"). The application should behave as a user expects dedicated hardware to behave.

2. The scheduler must allocate (nearly) 100% of available resources to ready tasks.

If a task is ready to run and some resource is partially unused, it will be made available to that task. For example, idle CPU time will be granted to a requesting task. If a task requests a resource that an earlier task reserved but is not using, the later task will be granted that resource if scheduling guarantees can still be met.

3. Quality of service modifications should be made in response to user requirements.

Assume that a system is overloaded because too much of one resource is required by the task set. A task must either be terminated (which is inconsistent with the first principles) or asked to shed load by providing a lower QOS. The decision as to which task(s) should be asked to shed load, and by how much they should be asked to degrade their service, should be based on user preferences. There must be a global understanding of how the task set is meeting the users' needs: QOS decisions should not be made (solely) in the context of a single thread.

3.3 Components and Control Flow of the Resource Distributor

The ETI Resource Distributor consists of three components: the Resource Manager, the Scheduler, and the Policy Box. Figure 2 shows these components and the communication paths among them.

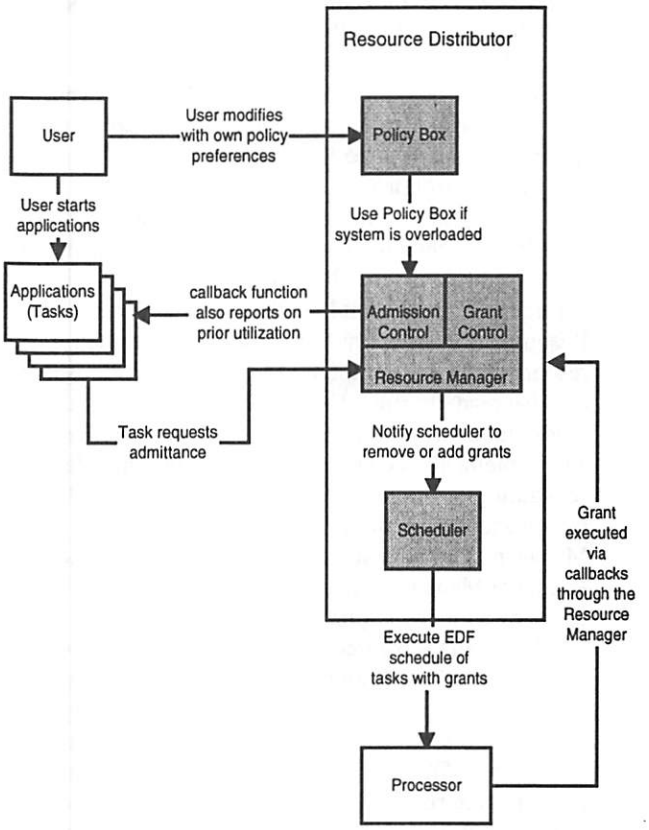


Figure 2: Components of the Resource Distributor.

The key insight that led to the design of the RD concerns the nature of application resource requirements: the QOS degradations that the applications can actually do are discrete. Performance levels are step-wise and known in advance by application writers. Resource allocations that do not map to a known service level in the application will result either in a missed deadline or in unused resources.

The *Resource Manager* allocates system resources to competing tasks. Allocation decisions are broken into two parts. The first, *admission control*, is the process of determining whether a thread can be admitted to the system. An admitted thread is guaranteed some non-zero amount of resources every period. Our approach requires that applications present a list of the load-shedding possibilities that they support at the time they request admittance. When the Resource Manager makes decisions, it thus has complete knowledge of the possibilities in the system.

The second part of the allocation decision, *grant control*, determines how much of each resource will be given to each thread (i.e., the *grant set*). The grant consists of a time period and an amount of resources that can be consumed in that time. For instance, a grant

might allocate 10 ms of CPU cycles in a 30 ms period. The grant is a guarantee to the thread that this much resource will be allocated to the thread in each period.

The work of actually ordering the threads to be run, and guaranteeing that granted resource allocations are delivered, is performed by the *Scheduler*. The scheduler makes no policy decisions: it simply cycles through the set of threads that has been established by the grant control process.

The Scheduler implements an Earliest Deadline First (EDF) schedule [Liu & Layland 73] of the tasks that have been specified by the Resource Manager. It enforces the grants that have been made by the Resource Manager, limiting applications to their resource allocation if there are other applications that are also ready to run. The Scheduler accepts grant modification information from the Resource Manager and implements the changes in a way that maintains the scheduling guarantees of the first principles; it also passes accounting information to the Resource Manager. The Scheduler communicates only with the Resource Manager -- never with the Policy Box, with users, or with any application.

When the Resource Manager is unable to give each application all the resources it has requested, it refers to the *Policy Box* to resolve the conflict. The Policy Box is a repository of information on how to make tradeoffs among the QOS possibilities for the different applications that are running. The Policy Box contains default settings, but these settings can be overridden by users.

3.4 Alternative Multimedia Schedulers

Existing multimedia schedulers address some of the requirements posed by a multimedia environment. However, none of them meets all the requirements or addresses all the first principles.

Other multimedia schedulers support both soft real-time and interactive tasks. Most are designed for use in a PC or workstation environment, where one or more users are doing conventional workstation tasks and there is multimedia activity. This differs from our environment in that we must supply tighter real-time guarantees to our multimedia tasks, and some of the interactive requirements of the workstation environment are absent.

All multimedia schedulers also acknowledge the need for QOS reductions in overload. When there are more demands for resources than can be met, the schedulers expect the applications to shed their load and recover gracefully.

Processor Capacity Reserves, from CMU [Mercer et al. 94], provides CPU reservations on a per-thread basis. Reservations are enforced, so a poorly behaved task cannot impinge on a well-behaved task with a reservation. Reservations can be passed between tasks in a multi-threaded environment. The actual scheduling algorithm used is Earliest Deadline First (EDF) and is based on the period for which a reservation is made.

The SMART Scheduler from Stanford [Nieh & Lam 97, Nieh & Lam 96] provides for the simultaneous execution of conventional and real-time tasks. It uses a modified best-effort scheduler. In underload, SMART meets all real-time constraints. In overload, conventional tasks continue to make progress, but real-time requirements are not necessarily met. Interactive tasks get good response, but low-priority threads may be denied service.

The Rialto system was developed at Microsoft Research [Jones et al. 97, Jones et al. 96, Jones et al. 95]. It combines resource reservation and constraint-based scheduling in an aggressive system that tries both to manage overload gracefully and to handle networked, independently authored, real-time environments. To better support cooperating processes, the scheduling algorithm uses minimum-laxity scheduling, with a concept of virtual time. The intent is to let cooperating, independently authored tasks reason about their real-time requirements.

None of these approaches meets all of the ETI RD's design requirements. Processor Capacity Reserves encourages applications to over-reserve, so the full processor may not be used. Both SMART and Rialto have approaches to avoid this, but they do not handle overload as required by our first principles. In SMART, overload is handled with fair-share scheduling, which conflicts with the discrete resource requirements of our applications. In Rialto, the nature of constraints causes the system to make policy decisions after a deadline may have already been missed. We discuss these implications in more detail later in the paper.

3.5 ETI Resource Distributor Benefits

The ETI RD is a real-time scheduler and resource manager that provides stronger guarantees than those provided by other soft real-time schedulers. It strictly adheres to the three first principles. As a result, it provides:

1. Better admissions control
2. Better resource allocation
3. Clear, practical separation of scheduling and QOS decision-making

Better Admissions Control

The RD guarantees its admissions: An admitted task is guaranteed that it will not miss a deadline. Most other systems cannot make this guarantee because of transient overload conditions. Examples include SMART, which does fair-share scheduling in overload,

and Rialto, which does not guarantee that a repeating constraint will be met in advance.

The RD does not reserve resources for a task that is not actually using them. Some systems, such as CMU's Processor Capacity Reserves, can provide guaranteed admission; however, these systems foster the over-reservation of resources so that deadlines can be met.

The RD also provides support for tasks that are not currently using resources but which cannot be denied admittance at some unspecified later time. These tasks, called *quiescent tasks*, are supported while still providing guaranteed admissions for non-quiescent tasks.

Better Resource Allocation

Unlike resource reservation schemes, the RD allocates 100% of available resources for ready tasks. The allocations are specifically tailored to the needs of the applications that are running. Unlike fair-share or best-effort schedulers, the RD allocates units of resources known to be useful to a thread. Resource allocations are made in quantum units; hence, resources are not allocated to a task that cannot meet its deadline. The actual resource allocations made are determined by the user. No accident of timing plays a part in the QOS provided by an application.

Clear, Practical Separation of Scheduling and QOS Decision-Making

The RD provides effective, reliable separation of scheduling and QOS decision-making. QOS decisions are always made in a global context, with reference to a user-defined Policy Box. The RD makes QOS decisions only during the time that has been allocated to the task requesting a global change in resource allocation. The RD's policy decisions are never made either in the context of a single application or when a thread will miss a deadline.

Alternative soft real-time systems deny users control of QOS decisions. There is usually some attempt to separate resource management, scheduling, and QOS decision making, but it does not go far enough. A system that makes any of its scheduling or resource allocation decisions in real-time fails in this regard when the system is in overload. Some of the literature acknowledges this clearly [Nieh & Lam 97], while other literature has not dealt explicitly with the issue of how to perform resource allocation and scheduling in overload conditions.

Most systems provide a failure notification to the application that is requesting resources. This independently authored application may then shed load. The problem with this approach is that the application that has just been denied service was selected by an accident of timing. The user might instead prefer that some other application degrade its service.

Alternative systems could assume a set of well-behaved applications, which would refer to a global third party to determine who should shed load. However, three problems remain. First, by the time the response returns from the third party, the deadline may no longer be reachable. Second, there is nothing in the literature that addresses how some other selected task, or the scheduler, would be informed that it is required to degrade its service. Third, even if another thread could be notified, it might either fail in the current frame or not degrade its service until later, causing other threads to miss their next deadlines.

4 RD Implementation

We now present the algorithms used by the Resource Manager, the Scheduler, and the Policy Box. We address some of the key issues in each area.

4.1 Resource Manager Algorithms

An application seeking real-time guarantees must access the Resource Manager to "request admittance." The application passes a resource list to the Resource Manager. The resource list is an ordered list of entries, each of which corresponds to one level of QOS that the application can provide.

Each entry contains a period and CPU requirement, both of which are specified in units of 27 MHz ticks. The reason for using the 27 MHz tick as a unit relates to clock synchronization issues and the MPEG tasks, as explained later in the paper. The minimum period is 500 μSec, and the maximum is 159 seconds. Each entry also contains a callback function associated with that level of QOS.

Table 1 shows the simplified format of a resource list. It omits several fields that manage resources other than CPU cycles on the MAP1000.

The "period(s)" selected by the applications are either naturally occurring, defined by some standard, or set by external events. For instance, MPEG needs to generate 30 frames per second, so a period of 1/30th of a second is needed. Expressed in terms of 27 MHz ticks, MPEG requests a period of 900,000 in its resource list. If the user in a PC environment had selected a display refresh rate of 72 Hz, a period of 375,000 ticks (27,000,000/72) would be used by 2D graphics. The period defines the start and end times of each time unit in which resources are allocated for this application. The start of period (n+1) is the same as the end of period (n).

The "CPU requirement" is a measure of the amount of CPU time the application requires during every period. If MPEG requires 1/3 of the CPU, it would pick a CPU requirement of 300,000 ticks. The "Rate" value

is computed as the CPU requirement/ period and represents the rate at which the resource list entry consumes CPU resources.

Period 27 MHz	CPU Req. 27 MHz	Rate (computed)	Function
$Period_{max}$	CPU Req. $_{max}$	CPU Req. $_{max}$ /Period	$Func_{max}$
$Period_j$	CPU Req. $_j$	...	$Func_j$
$Period_i$	CPU Req. $_i$	...	$Func_i$
...	...	...	...
$Period_{min}$	CPU Req. $_{min}$	CPU Req. $_{min}$ /Period	$Func_{min}$

Table 1: Simplified Format of a Resource List.

"Function" is the address of a routine that implements the level of QOS appropriate for the resource list entry. An application may have different functions for different entries. The scheduler upcalls to the function when the application has been granted the resources associated with the resource list entry.

Example resource lists for MPEG decoding and 3D graphics are shown in Tables 2 and 3, respectively. The MPEG application sheds load by selectively dropping more B frames. The 3D graphics application sheds load simply by making less progress on the same function. Neither of these examples necessarily reflects how real applications would shed load. The resource lists again are simplified.

Period	CPU Req.	Rate	Function
900,000	300,000	33.3 %	FullDecompress()
3,600,000	900,000	25.0 %	Drop_B_in_4()
2,700,000	600,000	22.2 %	Drop_B_in_3()
3,600,000	600,000	16.7 %	Drop_2B_in_4()

Table 2: Resource List for an MPEG Thread.

The resource list for a thread does not change if it is quiescent. A quiescent thread is not scheduled, because it is in a different mode. It is not necessary to add a null entry to the resource list for a quiescent thread.

When a task requests admittance, the Resource Manager performs two distinct tasks. The first, admission control, determines whether the application is allowed to run with scheduling guarantees: to be admitted to the domain of the ETI Resource Distributor. If the task is admitted, the Resource Manager performs a second task, determining the new grant set. The grant set determines which resources are made available to each admitted task.

Period	CPU Req.	Rate	Function
2,700,000	2,160,000	80%	Render3DFrame()
2,700,000	1,080,000	40%	Render3DFrame()
2,700,000	540,000	20%	Render3DFrame()
2,700,000	270,000	10%	Render3DFrame()

Table 3: Resource List for a 3D Graphics Thread.

A new thread is allowed to enter the system if and only if the sum of the minimal grants for all threads (runnable and quiescent) in the system can be simultaneously accommodated if the new thread is admitted. The admissions control test is expressed as:

$$\sum_{i=0}^{Runnable} Rate(\min)_i + \sum_{j=0}^{Quiescent} Rate(\min)_j \le 100\%$$

When a thread enters or leaves the system, or when a potentially quiescent thread changes state, the Resource Manager generates a new set of grants for all threads. The grant for a thread can increase or decrease at this time. The thread is informed indirectly, because the next period is started with a call to the function associated with the new grant. Because the Resource Manager ensures that the sum of grants does not exceed 100%, the scheduler need only enforce the grants to be able to use a simple EDF scheme to successfully schedule all threads. The calculation used to determine the new grant set is:

$$\sum_{i=0}^{Runnable} Rate\,(grant)_i \le 100\,\%$$

Because we admit a task only if the sum of the minimum resource list entries is less than the total resources available on the machine, we are assured that a legitimate grant set exists: at worst, all tasks receive their minimum grant. If possible, all tasks are given their maximum grant. However, if there are insufficient resources, the Policy Box is referenced to make trade-offs. An example grant set for three tasks is shown in Table 4.

	Period	CPU Req	Rate	Function
Modem	270,000	27,000	10%	Modem
3D	275,300	143,156	52%	Render 3DFrame
MPEG	810,000	270,000	33%	Full Decompress

Table 4: Grant Set for Three Threads: Modem, MPEG Decompression, and 3D Graphics.

The Resource Manager does its work in the context of the requesting application. Admission control is performed only when a new task tries to enter the system.

A new grant set is computed only when a task enters or leaves the system, when it changes its resource list, or when it enters or leaves the quiescent state.

One strength of the Resource Distributor is that the cost of computing a grant set is never paid using cycles that have already been committed to some other admitted task. By design, the costs of the Resource Manager cannot affect its ability to meet its scheduling guarantees. The Resource Manager decisions are made neither in interrupt mode nor when a deadline is in jeopardy. Coordinated communication of the new grant set to the Scheduler is done so that scheduling guarantees are maintained.

4.2 Scheduler Algorithms and Guarantees

The Scheduler implements an Earliest Deadline First (EDF) scheduling algorithm. The EDF algorithm is proven to be able to schedule any set of tasks for which there is a schedule. By not allocating more resources than there are, a set of tasks is guaranteed to be schedulable. EDF is extremely cheap to implement [Liu & Layland 73].

There are some rules governing the ability of an EDF scheduler to make scheduling guarantees.

1. *All tasks must be periodic.*

All tasks managed by the Resource Distributor are periodic tasks. Sporadic tasks are managed by a Sporadic Server, as discussed below.

2. *All tasks must be infinitely preemptible.*

While tasks are not infinitely preemptible, they are finely preemptible. In fact, to minimize context-switch overhead, we override the EDF policy when the overlap between two tasks is extremely small. If the currently executing thread has a distant deadline but only a small allocation of CPU time remaining, we complete it, even though another thread with a nearer deadline is runnable. The length of this override time is a function of the context-switch time.

3. *The period end is the period start.*

EDF does not support a separate start time and period start. This implies that an application must be willing to take its allocation at any point within its period.

4. *There can be no synchronization between tasks.*

The reason for this limitation is the same as the previous: a task must be willing to accept its allocation at any point in the period. If a task has blocked to synchronize, it might miss the (only) window in which it could be scheduled. Non-blocking synchronization is acceptable.

One implication of EDF is that the maximum guaranteed latency for a task is twice its period minus twice its CPU requirement. This occurs when the grant is delivered to an application at the beginning of one period and at the end of the subsequent period.

The Resource Manager notifies the Scheduler that a new grant is available. The next time there is unallocated CPU time, the Scheduler makes a callback to the Resource Manager to get the new grant information. By waiting for unallocated time to begin a new grant, we assure that adding a new task cannot affect the running of an already admitted task. The Scheduler is notified immediately that a grant should be removed or decreased, and the decrease occurs in the next period for the affected task.

The Scheduler maintains all tasks with grants on one of two queues: (1) The TimeRemaining queue, which contains all tasks that have unused CPU cycles allocated in this period, or (2) the TimeExpired queue, which contains all others. Both queues are ordered by deadline. A thread on the TimeExpired queue has either used its allocated CPU cycles for the period or indicated that it is done with its work for the period. A thread on the TimeExpired queue can also be on an Overtime-Requested queue if it ran out of time and still had more work to do.

On a context switch, the Scheduler takes the first thread off the TimeRemaining queue, if there is one. If no threads have time remaining but there are new grants, it calls back to the Resource Manager to get the new grant information. Finally, it takes the first thread off the OvertimeRequested queue. We always maintain at least one thread (the Idle thread) on this list.

The Scheduler sets a timer interrupt for the next context switch. This occurs at the earlier of: (1) the end of the grant for this thread for this period, or (2) the beginning of a new period for another thread whose next-period end precedes the period end for the thread about to run.

The next context switch is caused either by a timer interrupt or by the running task yielding the processor.

The ETI RD takes exactly those context switch interrupts required by the set of applications running on the system. We take (at least) twice as many interrupts as the shortest period in the system. If a task has a period of 5 ms, we switch context at least twice every 5 ms. The number of context switches can be minimized when tasks have the same period or periods that are multiples of each other, but this is an artificial restriction for most task sets.

Figure 3 shows a schedule for the grant set depicted in the example in Table 4. The EDF schedule preempts the MEPG and 3D Graphics tasks.

The ETI Resource Distributor makes the following scheduling guarantees to admitted tasks:

1. The task will receive a grant from the Resource List supplied by the application.
2. The grant will be delivered in each period.
3. Unless the task has the smallest CPU requirement in the system, it may be preempted each period.
4. The grant will not change mid-period.

5. The task will not be involuntarily terminated.

These guarantees are void for any period in which the task is blocked, but they will resume in the first full period in which the thread is not blocked.

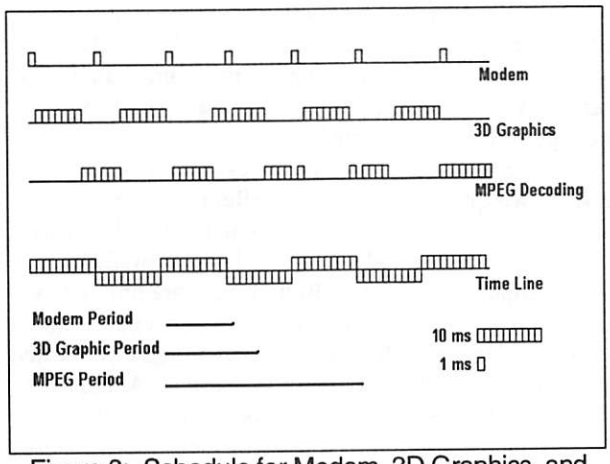


Figure 3: Schedule for Modem, 3D Graphics, and MPEG Decode.

4.3 Policy Box Algorithms

The Policy Box is a repository of information used to resolve conflicts when the system is overloaded. It is accessed by the Resource Manager when: (1) system requirements change, and (2) not all tasks can have their maximum resource list entry.

Because the Policy Box is always consulted when the system goes into overload, policy decisions are never made locally. Further, because decisions are made in the context of a task not yet admitted, they are not made when a deadline is about to be missed. The policy is not even affected by the order in which a set of threads is started. The cost of consulting the Policy Box is paid by the task requesting admittance. Therefore, no task will be denied service or miss a deadline as a result of the system going into overload.

The Policy Box has default policies supplied by the system designers, which can be overridden by users. For example, video should generally be degraded before audio, because most users are more sensitive to the quality of audio. However, in a loud environment, the clicks and pops of poor audio may be indistinguishable from ambient noise. In such an environment, where the user may rely more on video than audio, the quality requirements may be reversed. Alarms (with clicks and pops) are still needed, but the visuals must be current.

Each policy contains a relative ranking for its included threads. These rankings are used to compute which resource list entry each thread should receive. A simplified example Policy Box is shown in Table 5. In this example, there are four tasks known to the Policy Box. The Policy Box correlates a task name and Policy Box identifiers.

Policy ID	Rankings			
	Task 1	Task 2	Task 3	Task 4
1,2	10	85		
1,3	20		75	
1,4	10			85
1,2,3	10	50	35	
1,2,4	10	35		50
1,3,4	10		35	50
1,2,3,4	5	35	20	35

Table 5: Example Policy Box.

5 Additional Features

This section discusses important additional features and characteristics of the system. We describe our support for other task types, timing issues, the specifics of grant delivery, and preemptions.

5.1 Sporadic Tasks

Most tasks in our system are periodic and have real-time characteristics. Some tasks are neither periodic nor real-time. We call these *sporadic tasks*.

Sporadic tasks are managed by a Sporadic Server, which is itself a periodic task [Sprunt et al 89]. We provide an interface whereby any periodic task can "assign" its grant for a specific period of time to another (non-periodic) task. The Sporadic Server maintains a round-robin queue of the sporadic tasks in the system. When it is scheduled by the Scheduler, it looks for work to do on its queue. If there are sporadic tasks ready to run, the Sporadic Server assigns its grant for some fixed amount of time (currently 10 ms) to them. The Sporadic Server then returns to the Scheduler.

When the Scheduler selects a periodic task to run, it routinely checks to see if the task's grant has been assigned. If it has, the Scheduler runs the assigned-to thread instead. Resource bookkeeping is still done in the context of the periodic task. The assignment extends over multiple periods if more time is assigned than is available in a single period. When the grant is consumed, or when the sporadic thread blocks, the Scheduler returns to the periodic task.

Sporadic tasks can perform the same functions as periodic tasks. The only distinction is that there are no scheduling guarantees for a sporadic task. The performance of a sporadic task is a function of the amount of CPU time allocated to the Sporadic Server (which can be modified through the Policy Box) and the number of sporadic tasks.

5.2 Interrupt Tasks

The periodic model does not work well for low-latency, high-frequency tasks. Latency requirements of less than about 1 ms cannot be accommodated by periodic tasks running under the ETI Resource Distributor. (Recall that the best guaranteed latency is two times the period minus two times the CPU cycle allocation.) Generally, these tasks are triggered by an interrupt and must be serviced by an interrupt handler.

Because these tasks do not come under the purview of the Resource Distributor, their resources are not taken into account. We reserve a small percentage of the processor for handling interrupts.[3] Tradeoffs must be made between keeping this number small to avoid wasted resources and making it large enough that interrupts do not conflict with the deadlines for admitted tasks.

5.3 Quiescent Tasks

A quiescent task is one that is not currently using any resources, but which cannot be denied admittance when ready to run. One example is a cool-down task. If the processor is overheating, the operating system is notified and is expected to cool down the processor. It does this by running a no-op loop that switches fewer transistors. The cool-down task does not need to use 100% of the processor (because if it did, it might make more sense to shut down the system); however, it does need some percentage, depending on the extent of overheating. Until the processor has overheated (if ever), we do not want to reserve resources for this task. However, if the processor has overheated, we must run the cool-down task.

An obvious, simple way to handle this situation is to terminate some other task that is running. However, this approach violates our scheduling guarantee.

A better way is to incorporate the quiescent task into the calculation for admissions control and to ignore it for calculating the grant set. With this approach, when the task ceases to be quiescent, we are guaranteed a grant set for all admitted tasks: at worst, all tasks receive their minimum resource list entry. However, while the task is quiescent, the resources it would otherwise use are allocated to other tasks, enabling them to provide a higher level of QOS.

[3] On the MAP1000, the Data Streamer greatly reduces the number of interrupts taken. Most tasks are double or triple buffered. In period (n+1), they may be outputting the data generated in period (n). This, combined with the fact that the Data Streamer can be programmed to do flow control, reduces the number of I/O interrupts in the non-error case to near zero.

An example is a telephone-answering modem task. Imagine a PC environment where the user is studying multimedia data from a DVD. The user is waiting for a teleconferencing connection to be established so that the multimedia data can be discussed. Until the telephone call occurs, the full resources of the machine should be dedicated to the DVD. Afterwards, the modem, teleconferencing, and DVD software must share resources, and the DVD may have to shed load. Our Resource Distributor lets the user start these applications in any order. The desired global policy decisions will be made, the correct load shedding performed, and the telephone answered promptly.

5.4 Clock Synchronization Issues

The periods of applications are frequently defined by external clocks. For instance, MPEG applications are tied to the TCI clock, which runs at 27 MHz [ISO 96]. The TCI clock is routinely modified by the MPEG system software to stay synchronized to an incoming MPEG data stream. In other words, it is known that the MPEG clock will drift with respect to any other clock. The system clock on the MAP1000 is 200 MHz.

Clocks driven by different crystals can drift with respect to each other. One can be running slightly faster, one slightly slower. The TCI clock can do both. Sometimes it drifts faster, sometimes slower, depending on the source of the MPEG input stream.

The drifting of two clocks with respect to each other causes problems for periodic threads whose period is externally defined. For example, imagine that a user has selected a 100 Hz display refresh rate. Every 10 ms the MAP1000 needs to have an image ready to display, and every 10 ms the Display Refresh Controller (DRC) picks it up. Since the image-generating application on the MAP1000 and the DRC are controlled by different clocks, they can drift with respect to each other. In time, one of them can get an entire frame ahead of or behind the other. At this point, either an entire frame is dropped, or a frame is displayed in duplicate.

Unlike the case for many applications, the DRC can ignore this problem at a fairly small cost. It is commonly the case that the same screen is displayed multiple times, since the MPEG frame rate is typically less than the refresh rate of a non-interlaced RGB computer monitor. Further, there is unlikely to be anything unique about a single display frame that would make its loss seriously reduce quality. A more significant problem is tearing: if the DRC displayed half of one frame and half of the next, the user could detect a quality degradation. This problem is common with most schedulers. It is usually avoided by changing the pointer to the data to be displayed only when it is complete. For the DRC, clock synchronization issues are relatively easy to manage.

However, some other applications have significant problems with clock synchronization. One example is an MPEG decoder. The MPEG data stream is received live, at 30 frames per second. The stream is compressed, and not all the delivered frames are the same type. An MPEG stream has I, B, and P frames. The I frames are Initial frames without temporal prediction: they can be decoded in isolation. The compressed P frames are encoded as the difference from the previous I or P frame. The compressed B frames are encoded as the difference relative to both the preceding and following I or P frames. The significance of losing a B frame is small – one frame is not displayed. However, if an I frame is lost, a perfect picture cannot be displayed again until the next I frame is received, which is typically every 15 frames or half-second. A half-second loss of video is noticeable and unacceptable. Therefore, for MPEG, if an I frame is lost, the QOS would be inadequate.

Because the consequences of losing an I frame are unacceptable, we have partially finessed the problem of staying synchronized with the TCI clock by using the TCI clock for scheduling. This means that the first MPEG transport stream does not need to worry about TCI synchronization. However, a second MPEG thread using a different TCI transport stream, and any other application that wants to stay synchronized with an external clock, must do so in software.

We provide an interface (InsertIdleCycles) that can be used to postpone the start of the next period for a task by an arbitrary number of 27 MHz ticks. Postponing the period cannot jeopardize the scheduling guarantees to other tasks, but pulling it in would, so the interface cannot be used to pull in the period start. This interface can be used both to control the effects of drift from clock skew and to get into phase with a clock. The application must read both the TCI and the external clock at some interval. The difference between the external clock readings is determined. From that, the expected difference in the TCI clock is computed. The actual difference in the TCI clock readings can be used to calculate the skew.

5.5 Semantics for Delivering a Grant

When a task receives a grant, we make a callback to the function named in the resource list entry. The stack is cleared before the call, and the calling arguments include whether the previous call completed, the sum of the resources used in the previous call, and an indicator of which grant has been assigned for this period. This is how the initial grant for an admitted task is always delivered.

We offer both callback and return semantics for all periods past the initial one. For truly periodic tasks (e.g., MPEG, modem, audio), a callback is generated at the beginning of every new period. The work of the previous period is complete, and the exact same function is performed on the set of data in this new period. For tasks such as 2D and 3D graphics, which are not as tightly tied to their periods, the state between periods should be retained, and the application should continue where it left off. These tasks use return semantics. Note that all tasks use return semantics when they have been preempted in the middle of their grant for the period; callback semantics apply only at the beginning of a new period.

A task must be prepared to receive a new grant in any period. For tasks using callback semantics, this is trivial. For those using return semantics, some clean-up operations may be required. Depending on the differences between the old and new grants, the task may even prefer to use return semantics on the new grant.

An example is the 3D graphics task, which has multiple resource list entries that use the same function. On the MAP1000, the 3D graphics application has some resource list entries that use the video scaler component of the FFU, and some that do not. If the grant change involves either acquiring or losing access to this unit, then the 3D graphics task needs to use callback semantics to get the grant started after some clean-up. If the access to the FFU does not change, it uses return semantics.

We provide a filter callback option for a task that uses return semantics when its grant changes. If the task has registered a filter callback, we call it instead of either returning or using the new grant's callback function. The task does whatever cleanup is necessary and then returns an indicator as to whether it would prefer return or callback semantics for this one call.

5.6 Preemptions and Real-time Applications

As long as the tasks in a system have differing periods and CPU requirements, it will be necessary to preempt the tasks with longer CPU requirements and periods. Preemptions are expensive and disruptive. Besides the context switch overhead, the cache state may also be lost.[4]

The majority of tasks we run are double or triple buffered. At the end of a buffer, some data will even be jettisoned. The best time to preempt a task is when it has finished handling one buffer and before it starts on the next. The "buffers" may be relatively small; they do not correspond to an entire frame, for instance, but perhaps to an MPEG macroblock, which is 384 bytes.

[4] Our chip has additional complications with the Data Streamer and the FFU. At any given point, both the Data Streamer and the FFU are likely to be performing long-running operations on behalf of the current task.

To minimize the cost of preemptions, we provide a mechanism with which a well-behaved task can perform *controlled preemptions*. Otherwise, it is preempted as usual.

We distinguish between involuntary and voluntary preemptions. An *involuntary preemption* occurs in a normal context switch, when the process is being given to another task. A *voluntary preemption* occurs when a task blocks, for example, on synchronization or I/O. A voluntary preemption also occurs when the task volunteers to yield the processor because the Scheduler must do a context switch.

To perform controlled preemptions, the task notifies the Resource Manager of its intention, and the Resource Manager notifies the Scheduler. When the Scheduler needs to preempt the task, it sets a marker indicating that a context switch is needed, notifies the task, and sets a timer interrupt for a grace period. Before the grace period expires, the task must notice that it is in the grace period and voluntarily preempt itself by yielding the processor. If the grace period expires before the task yields, it is involuntarily preempted.

The task can specify a local address at which it would like to be notified when a context switch is needed. By doing so, the task avoids the system call needed to determine if a preemption is required and may avoid additional cache misses. The grace period is quite short – on the order of a couple hundred µSec.

The grace period effectively lets one task run into the time allotted to another. The task will be charged for the resources it uses in the grace period; however, the other task is still postponed. Therefore, it is critical to keep the grace period as small as possible. On the other hand, the more often that applications have to check for preemption, the more constrained their coding, because they must be prepared to yield the processor. It remains a matter of further study to determine the optimal grace period length.

If a task is doing voluntary preemptions and fails to yield in the grace period, it is involuntary preempted. When next run, it is sent an exception callback, enabling it to clean up.

6 Performance

This section presents performance data for the costs involved in context switches, admissions, grant set determination, managing preemption and scheduling. Most costs are incurred in the context of a task either requesting admittance or significantly changing its state; the run-time costs are relatively small. All performance numbers reported in these sections were acquired on a cycle-accurate simulator.

6.1 Context-Switch Costs

One advantage of the ETI Resource Distributor is that preemptions are taken only when required for correctness of scheduling guarantees. The number of context switches depends on the number of applications, and the size of their periods and CPU requirements. Rialto reduces the number of context switches by enforcing the rule that all applications have periods that are even multiples of each other [Jones et al. 97]; we support any period length in range.

A context switch on the MAP1000 incurs the cost of saving and restoring as many as two banks of 64 32-bit registers. In our calling standard, most registers are caller-saved; therefore, for a synchronous (voluntary) context switch, only 14 32-bit registers (times two banks) must be saved. There are another 64 32-bit system registers that must be saved on an involuntary context switch.[5]

On a 200 MHz chip, a fully synchronous, voluntary context switch takes a minimum, median, and average of 11.5, 18.3, and 20.7 µSec. A fully involuntary context switch takes a minimum, median and average of 16.9, 28.2, and 35.0 µSec.

On a highly tuned system running an MPEG video decoder and AC3 audio, we might expect about 300 context switches per second (i.e., 60 each for the MPEG decoder and AC3 audio and for each of their data management threads, and another 30 for the Sporadic Server). Of these, 120 will be synchronous context switches, for a total of 2196 µSec. The remaining 180 context switches are asynchronous, at a mean cost of 28.2, for a total of 5076 µSec. For this load, we would expect a total context-switch cost of about 0.7% of the CPU.

Threads that have the same period do not preempt each other. If threads have different periods, the one with the shorter period can preempt the one with the longer.

6.2 Admissions Control Cost

Admissions control is computed in constant time. A running sum of the resources used for each thread's minimum resource list entry is maintained. When a new thread requests admittance, the resources of its minimum resource list entry are added to the running total and compared to what is available on the system. On a 200 MHz system, the admissions control process takes between 150 and 200 µSec.

[5] The cost of saving Data Streamer and FFU state is not considered part of the context-switch cost. For a well-behaved application, the cost is zero.

6.3 Determining a Grant Set Cost

The cost of determining a grant set is a function of: (1) whether the system is in overload, and (2) the number of threads admitted to the system. If the system is not in overload, we first make an O(1) determination as to whether every thread can have the resources requested in its maximum resource list entry. If so, we are done.

If the system is in overload, the computation becomes more complex. When the Resource Manager finds that not all threads can have their maximum, it asks the Policy Box for a policy for the set of admitted, non-quiescent threads. The Policy Box searches its database for a matching policy. If it does not find one, the current implementation invents a policy in which each of N threads receives 1/Nth of the resources, and an arbitrary thread is given control of exclusive resources.

Once it has received a policy, the Resource Manager correlates the policy received with the actual resource list entries of the threads. Our current implementation for this step is O(N). We iterate through each thread, noting the resource list entries just above and below the QOS specified by the policy. If the sum of the entries that were above the policy-specified QOS ratings fits, we are done. Otherwise, we walk through once more, turning higher entries into lower entries. This process will converge in a single pass, because only policies that fit are allowed by the Policy Box. We make a third pass if substantial resources remain unused after the second pass, looking for a thread that can use these otherwise unallocated resources.

6.4 Managing Preemption Cost

Threads that want to do controlled, voluntary preemptions make a call to the Resource Manager, giving a local address in which the notification should be placed. The cost of a managed preemption is potentially much less than the cost of an involuntary context switch. The application writer controls what information is in the caches and the states of the FFU and Data Streamer.

There are two incremental costs to doing controlled preemptions. First, the thread must periodically check to see if it is in a grace period. It must do this frequently enough so that it can get into a safe state and yield the processor before the grace period expires. On the MAP1000, the process of checking to see if a thread is in the grace period is essentially free. On any architecture with a functional unit that is not 100% utilized, an otherwise idle cycle on that unit can be used to check if preemption is required.

The second, and higher, incremental cost of controlled preemptions is incurred in the operating system, where we must notify the thread that it is in a grace period. When the operating system receives a timer interrupt, we check to see if the running thread is eligible for a grace period. If so, we set the location specified by the thread, reset the timer interrupt for the grace period, and continue the interrupted thread.

6.5 Scheduler Effectiveness

The ETI Resource Distributor effectively schedules a set of threads. We are currently running simultaneous sets of MPEG, AC3, and 3D graphics. There are a number of ancillary support threads for data management, for the Display Refresh Controller, etc.

The following data are from a test run with four periodic threads in addition to the Sporadic Server. Each is running with a period of 1/30th of a second, and each has a maximum CPU requirements of 13, 2, 3 and 3 ms, respectively. The thread with the largest requirement never reports that it has finished its work for the period. This set of threads does not overload the system.

Figure 4 shows the schedule one-third of a second into the run. The two data control threads (8 and 10) are waiting for more data from the producers. Producer thread 7 receives the unused time (shown in the lighter lines) but is preempted when a new period begins; it then receives its guaranteed allocation (shown in the darker lines). The other producer thread (9) completes its work each period.

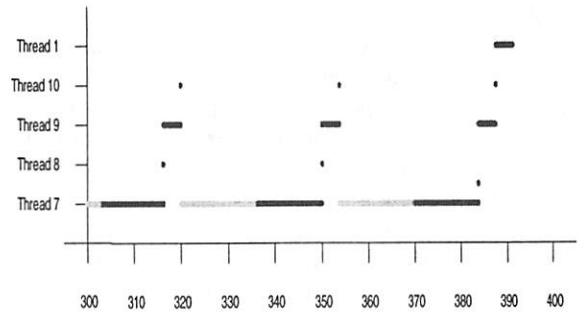


Figure 4: Schedule for Five Threads from 300-400 Ms. (Dark Lines show guaranteed allocations; light lines show allocations of unreserved time.)

This example shows a bug in the application: the data management threads should block, waiting for the data to become available. The context switches to the data management threads could be avoided when no data is available. The producer threads could set an event when data is available, and the data management threads would regain their scheduling guarantees in the following period.

The next test shows five threads in addition to the Sporadic Server, each of which has nine entries in its resource list. Each resource list entry has a period of 10 ms, and the nine entries range from requiring 90% to 10% of the CPU. Because there are no policies for

these threads, we expect the Policy Box to make up a policy that evenly divides the resources among the available threads, including the Sporadic Server. The Sporadic Server requires only 1% every 100 ms, but it is the only thread that indicates it has work to do at the end of each period; the other threads all yield when preemption is required.

Each of the threads is started in turn, with a wait of 20 ms between each thread start. Because the period for each thread (except the Sporadic Server) is 10 ms, we expect each thread to be scheduled every 10 ms. As each new thread is admitted, we expect the CPU allocation for all previously admitted threads to be reduced. Since we reserve 4% of the processor for interrupt processing, and there are no interrupts other than those for the timer in this run, we expect the Sporadic Server to run at least every 10 ms. We also expect that each new thread will receive its first grant in a period that would otherwise have been given to the Sporadic Server as unallocated time.

Table 6 shows the resource list for each of the threads 2-6. Figure 5 shows that the run has behaved as expected. Thread 2 (the first periodic thread admitted after the Sporadic Server) begins with an allocation of 9 ms out of 10. It then drops to 4 ms when one thread is added, to 3 ms when there are three threads, and to 2 ms when there are four or five threads running in addition to the Sporadic Server. Each thread's allocations are received 10 ms apart, because the period does not change.

7 Conclusions and Future Directions

The ETI Resource Distributor is unique in its ability to provide scheduling guarantees to a dynamic set of applications without wasting resources. Like other soft real-time systems, we support the dynamic creation of tasks and the possibility of overload. Like hard real-time systems, we provide guarantees regarding the resources that will be made available to a task and the timeframe in which they will be provided. By designing for the step-wise nature of resource usage in multimedia applications, we support load shedding in a way that meets the user's performance requirements.

Period	CPU Req.	Rate	Function
270,000	243,000	90%	BusyLoop()
270,000	216,000	80%	BusyLoop()
270,000	189,000	70%	BusyLoop()
270,000	162,000	60%	BusyLoop()
270,000	135,000	50%	BusyLoop()
270,000	108,000	40%	BusyLoop()
270,000	81,000	30%	BusyLoop()
270,000	54,000	20%	BusyLoop()
270,000	27,000	10%	BusyLoop()

Table 6: Resource List for Threads 2-6.

We are also unique in providing practical control of QOS policy decisions, even in the face of a dynamic task set with firm real-time requirements. Our system truly separates policy decisions from accidents of timing, task creation order, and other inadvertent influences. Even with a dynamic task set, we provide exactly the policy that is desired.

This system has a low run-time cost, and provides reliable QOS, and guaranteed scheduling. It guarantees liveness for conventional tasks, and also supports the Quiescent task model for tasks that cannot be denied service.

There remain areas for additional research. First is the better integration of more resources. Our implementation supports the CPU, the FFU, and the Data Streamer. However, we do not specifically manage

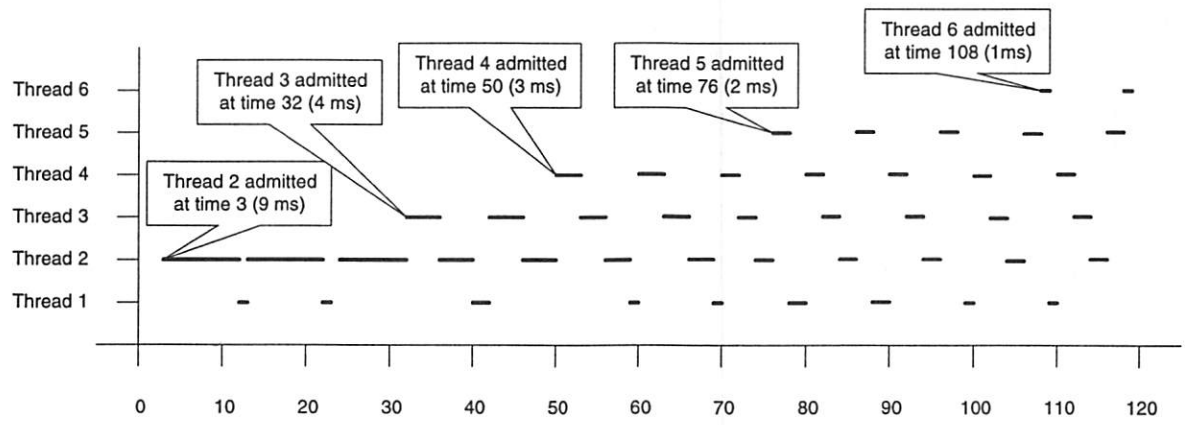


Figure 5: Schedule for Five Load-Shedding Threads Plus the Sporadic Server.

bandwidth as a resource, but we will need to do so when the number of applications using the Data Streamer increases. It is possible that memory should also be incorporated. However, every additional resource increases the complexity of the algorithms.

A second area for more investigation is the Policy Box. Ours can be accessed by applications, the user, and the operating system. There are open issues around policy modifications: when is it reasonable to change the Policy Box, and when should the modification(s) occur to avoid affecting current scheduling guarantees?

Currently, policies are specified as relative rates. There are two problems with this. First, other resources are not integrated. Second, the correlation between the "rates" of the Policy Box are not theoretically well matched to the rates of the applications. Future research will find a better way of expressing possibilities in the Policy Box without limiting the range of resources used by the applications.

8 Acknowledgements

I wish to thank my editor, Sandy Kaplan, my shepherd, Sape Mullender, and the anonymous reviewers who made many helpful comments on earlier drafts. Thanks to my colleagues at Equator Technologies, Inc. who provided endless support for this work.

Bibliography

[Basoglu et al 99] Chris Basoglu, Robert Gove, Keiji Kojima, and John O'Donnell. Single-Chip Processor Media Applications: The MAP1000™. In *Int J Imaging Syst Technol,* 10, 1999, in press.

[Helander & Forin. 98] Johannes Helander and Alessandro Forin. MMLite: A Highly Componentized System Architecture. In *Proceedings of the Eighth ACM SIGOPS European Workshop on Support for Composing Distributed Applications,* Sintra, Portugal, Sep. 1998.

[ISO 96] ISO/IEC International Standard 13818-1, Information Technology – Generic coding of moving pictures and associated audio information: *Systems*. 1996.

[Jeffay & Bennet. 95] Kevin Jeffay and David Bennett. A Rate-Based Execution Abstraction For Multimedia Computing. In *Proceedings of the Fifth International Workshop on Network and Operating Systems Support for Digital Audio and Video,* April, 1995.

[Jones et al. 95] Michael B. Jones, Paul J. Leach, Richard P. Draves, Joseph S. Barrera, III. Modular Real-Time Resource Management in the Rialto Operating System. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, pp. 12-17. IEEE Computer Society, WA, May, 1995.

[Jones et al. 96] Michael B. Jones, Joseph S. Barrera, III, Alessandro Forin, Paul J. Leach, Daniela Rosu, Marcel-Catalin Rosu. An Overview of the Rialto Real-Time Architecture. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, pp. 249-256, Sep. 1996.

[Jones et al. 97] Michael B. Jones, Daniela Rosu, Marcel-Catalin Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, Oct. 1997.

[Liu & Layland 73] C.L.Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. In *Journal of the ACM*, vol.20, pp. 46-61, Jan. 1973.

[Mercer et al. 94] Clifford W. Mercer, Stefan Savage, Hideyuki Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[Nieh & Lam 97] Jason Nieh and Monica S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, Oct. 1997.

[Nieh & Lam 96] Jason Nieh and Monica S. Lam. The Design of SMART: A Scheduler for Multimedia Applications. Technical Report CSL-TR-96-697, Compute Systems Laboratory, Stanford University, June 1996.

[Sprunt et al. 89] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic Task Scheduling for Hard-Real-Time Systems. In *The Journal of Real Time Systems 1*, 1 Nov. 1989.

A Feedback-driven Proportion Allocator for Real-Rate Scheduling

David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee,
Calton Pu, and Jonathan Walpole
Department of Computer Science and Engineering
Oregon Graduate Institute

Abstract

In this paper we propose changing the decades-old practice of allocating CPU to threads based on priority to a scheme based on proportion and period. Our scheme allocates to each thread a percentage of CPU cycles over a period of time, and uses a feedback-based adaptive scheduler to assign automatically both proportion and period. Applications with known requirements, such as isochronous software devices, can bypass the adaptive scheduler by specifying their desired proportion and/or period. As a result, our scheme provides reservations to applications that need them, and the benefits of proportion and period to those that do not. Adaptive scheduling using proportion and period has several distinct benefits over either fixed or adaptive priority based schemes: finer grain control of allocation, lower variance in the amount of cycles allocated to a thread, and avoidance of accidental priority inversion and starvation, including defense against denial-of-service attacks. This paper describes our design of an adaptive controller and proportion-period scheduler, its implementation in Linux, and presents experimental validation of our approach.

1 Introduction

CPU scheduling in conventional general purpose operating systems performs poorly for real-rate applications, applications with specific rate or throughput requirements in which the rate is driven by real-world demands. Examples of real-rate applications are software modems, web servers, speech recognition, and multimedia players. These kinds of applications are becoming increasingly popular, which warrants revisiting the issue of scheduling. The reason for the poor performance is that most general purpose operating systems use priority-based scheduling, which is inflexible and not suited to fine-grain resource allocation. Real-time operating systems have offered another approach based on proportion and period. In this approach threads are assigned a portion of the CPU over a period of time, where the correct portion and period are analytically determined by human experts. However, reservation-based scheduling has yet to be widely accepted for general purpose systems because of the difficulty of correctly estimating a thread's required portion and period.

In this paper we propose a technique to dynamically estimate the proportion and period needed by a particular job based on observations of its progress. As a result, our system can offer the benefits of proportional scheduling without requiring the use of reservations. With these estimates, the system can assign the appropriate proportion and period to a job's thread(s), alleviating the need for input from human experts. Our technique is based on feedback, so the proportions and periods assigned to threads change dynamically and automatically as the resource requirements of the threads change. Given a sufficiently general, responsive, stable, and accurate estimator of progress, we can replace the priority-based schedulers of the past with schedulers based on proportion and period, and thus avoid the drawbacks associated with priority-based scheduling.

This project was supported in part by DARPA contracts/grants N66001-97-C-8522, N66001-97-C-8523, and F19628-95-C-0193, and by Tektronix, Inc. and Intel Corporation.

The fundamental problem with priority-based scheduling is that knowledge of a job's priority by itself is not sufficient to allocate resources to the job properly. For example, one cannot express dependencies between jobs using priorities, or specify how to share resources between jobs with different priorities. As a result, priority-based schemes have several potential problems, including starvation, priority inversion, and lack of fine-grain allocation. Use of adaptive mechanisms like the multi-level feedback scheduler[3] alleviate some of these problems, but introduce new ones as the recent deployment of fixed real-time priorities in systems such as Linux and Windows NT can attest.

Our approach avoids these drawbacks by using a controller that assigns proportion and period based on estimations of a thread's progress. It avoids starvation by ensuring that every job in the system is assigned a non-zero percentage of the CPU. It avoids priority inversion by allocating CPU based on need as measured by progress, rather than on priority. It provides fine-grain control since threads can request specific portions of the CPU, e.g., assigning 60% of the CPU to thread X and 40% to thread Y.

The key enabling technology to our approach is a feedback-based controller that assigns proportion and period to threads based on measurements of their progress. For example, the progress of a producer or consumer of a bounded buffer can be estimated by the fill level of the buffer. If it is full, the consumer is falling behind and needs more CPU, whereas the producer has been making too much progress and has spare CPU to offer. In cases where progress cannot be directly measured, we provide heuristics designed to provide reasonable performance. For example, the scheduler can give interactive jobs reasonable performance by assigning them a small period and estimating their proportion by measuring the amount of time they typically run before blocking.

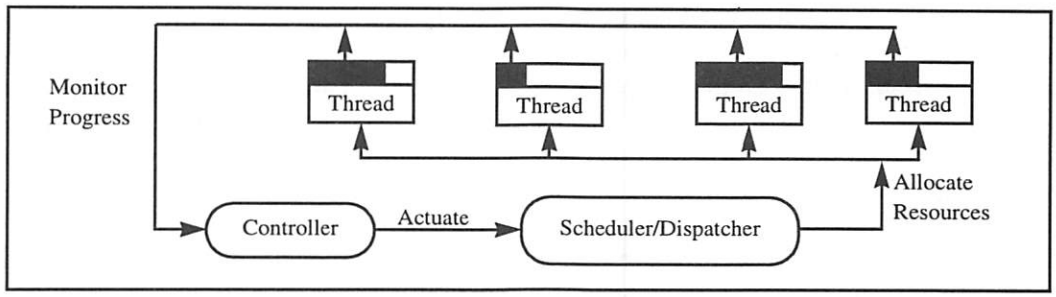
The remainder of this paper describes our approach in more detail. Section 2 motivates the need for adaptive proportion/period schedulers. Section 3 presents our solution, including a description of our implementation. Section 4 discusses implications of our solution, and presents experimental measurements of our prototype. Section 5 describes similar approaches to the question of scheduling.

2 Motivation

The limitations of priority-based scheduling were graphically demonstrated to the world recently when NASA's Mars Pathfinder robot experienced repeated resets due to priority inversion [13]. Occasionally, a high priority task was blocked waiting for a mutex held by a low priority task. Unfortunately, the low priority task was starved for CPU by several other tasks with medium priority. Eventually, the system would detect that the high priority task was missing deadlines and would reset itself. More insidious than the problem itself is the difficulty of finding the bug when it occurs. In this case, the mutex was buried under several layers of abstraction; no reasonable amount of code inspection would have discovered the bug. Fortunately, a combination of good engineering, run-time debugging support, and the fact that a mutex was the source of the inversion helped NASA engineers to correct the bug [12][17].

The problems of priority inversion and starvation occur because priorities alone are not expressive enough to capture all desired relationships between jobs. As a result, priority-based schemes are forced to use kludges to compensate, such as passing priorities through mutexes or decreasing the priority of CPU-bound jobs. These mechanisms have worked well in the past, but they have untoward side-effects.

For example, to ensure that the kernel allocates sufficient CPU to an important CPU-bound job running on Unix, one could *nice* it. However, as it continues to use its time-slice the kernel will automatically reduce its priority until it is running at or below the priorities of less important jobs. Alternatively, one could assign it a fixed real-time priority which is higher than the normal priorities, guaranteeing that it will run. Unfortunately, it will then run to the exclusion of all jobs in the system with lower priority. Consider a job running at a (fixed) real-time priority that spin-waits on user input. Since the X server typically runs at a lower priority than the real-time thread, it will be unable to generate the input for which the thread is spinning, and the system will livelock. Note that the solution used by the Mars Pathfinder of passing priority through mutexes[18] will not help in this situation.



This diagram shows the rough architecture of our scheduler. A feedback controller monitors the rate of progress of job threads, and calculates new proportions and periods based on the results. Actuation involves setting the proportion and period for the threads. The scheduler is a standard proportion/period reservation-based scheduler. The controller's execution period and the dispatch period can be different.

Figure 1: Diagram of Closed-loop Control

3 Our Solution

Our solution is based on the notion of *progress*. Ideally, resource allocation should ensure that every job maintains a sufficient rate of progress towards completing its tasks. Allocating more CPU than is needed will be wasted, whereas allocating less than is needed will delay the job. In essence, our solution monitors the progress of jobs and increases or decreases the allocation of CPU to those jobs as needed. In our terminology, a job is a collection of cooperating threads that may or may not be contained in the same process.

Figure 1 shows the high-level architecture of our design. The scheduler dispatches threads in order to ensure that they receive their assigned proportion of the CPU during their period. A controller periodically monitors the progress made by the threads, and adjusts each job's proportion automatically. We call this adjustment *actuation* or *adaptation,* since it involves tuning the system's behavior in the same sense that an automatic cruise control adjusts the speed of a car by adjusting its throttle. Readers should note that the diagram resembles a classic closed-loop, or feedback, controlled system. This dynamic adaptation controlled by feedback is necessary because the needs of jobs, and the composition of jobs running on the system vary with time. The following subsections address each of the key points in the architecture.

3.1 The Reservation Scheduler

Our scheduler is a standard *"reservation-based"* scheduler that allocates CPU to threads based on two attributes: proportion and period. The proportion is a percentage, specified in parts-per-thousand, of the duration of the period during which the application should get the CPU, and the period is the time interval, specified in milliseconds, over which the allocation must be given. For example, if one thread has been given a proportion of 50 out of 1000 (5%) and a period of 30 milliseconds, it should be able to run up to 1.5 milliseconds every 30 milliseconds.

Intuitively, the period defines a repeating deadline. To meet the deadline, the application must perform some amount of work. Hence, to satisfy the application the scheduler must allocate sufficient CPU cycles, which in our case is the proportion times the period times the CPU's clock rate. If the scheduler cannot allocate the appropriate amount of time to the thread, the thread is said to have missed a deadline.

An advantage of reservation-based scheduling[1] (RBS) is that one can easily detect overload by summing the proportions: a sum greater than or equal to one indicates the CPU is oversubscribed. If the scheduler is conservative, it can reserve some capacity by setting the overload threshold to less than 1. For example, one might wish to reserve capacity to cover the overhead of scheduling and interrupt handling.

1. Our use of the term "reservation" is somewhat loose, since we do not need strict guarantees from the scheduler. As a result, a good enough best-effort proportion/period scheduler would suffice.

Upon reaching overload, the scheduler has several choices. First, it can perform admission control by rejecting or cancelling jobs so that the resulting load is less than 1. Second, it can raise quality exceptions to notify the jobs of the overload and renegotiate the proportions so that they sum to no more than the cutoff threshold. Third, it can automatically scale back the allocation to jobs using some policy such as fair share or weighted fair share. In our system, these mechanisms are implemented by the controller, and are discussed below.

We have implemented a RBS scheduler in the Linux 2.0.35 kernel by adding a new scheduling policy that implements rate-monotonic scheduling (RMS)[14] using Linux's basic scheduling mechanisms[2]. Linux implements a variant of the classic multi-level feedback scheduling that uses one run-queue, and selects the thread to run next based on a thread property called *goodness*. At dispatch, i.e. when deciding which thread to run next, Linux selects the thread with the highest goodness on the run queue. If all threads on the run-queue have a zero goodness value, Linux recalculates goodness for all threads in the system. Each thread has a scheduling policy that is used by Linux for calculating goodness. Our policy calculates goodness to ensure that threads it controls have higher goodness than jobs under other policies, and that jobs with shorter periods have higher goodness values. When a thread has used its allocation for its period, it is put to sleep until its next period begins. Because enforcement of our RBS scheduling policy can only be made at dispatch time, we call this low-level scheduler the *dispatcher*, and the time between dispatches the *dispatch interval*. The interval is bounded above by the timer interval, which we have set to be 1 millisecond for our prototype. The key features of this prototype RBS are very low overhead to change proportion and period, and fine-grain control over proportion and period values. We could equally well have used other RBS mechanisms such as SMaRT [15], Rialto [11], or BERT [1] had one been available on our platform.

3.2 Monitoring Progress

The novelty of our approach lies in the estimation of progress as the means of controlling the CPU allocation. Unfortunately, estimating an application's progress is tricky, especially given the opaque interface between the application and the operating system. Good engineering practice tells us that the operating system and application implementations should be kept separate in order that the operating system be general and the application be portable.

Our solution to this problem is based on the notion of *symbiotic interfaces*, which link application semantics to system metrics such as progress. For example, consider two applications with a producer/consumer relationship using a shared queue to communicate. A symbiotic interface that implements this queue creates a linkage to the kernel by exposing the buffer's fill-level, size, and the role of each thread (producer or consumer) to the system. With this information, the kernel can estimate the progress of the producer and consumer by monitoring the queue fill level. As the queue becomes full (the fill-level approaches the maximum amount of buffering in the queue), the kernel can infer that the consumer is running behind and needs more CPU and that the producer is running ahead and needs less CPU. Similarly, when the queue becomes empty the kernel can infer the producer needs more CPU and the consumer less. This analysis can be extended to deal with pipelines of threads by pairwise comparison. Over time, the feedback controller will reach equilibrium in steady-state provided the design is stable.

Our solution is to define suitable symbiotic interfaces for each interesting class of application, listed below. Given an interface, we can build a monitor that periodically samples the progress of the application, and feeds that information to the controller.

- Producer/Consumer:

  The applications use some form of bounded buffer to communicate, such as a shared-memory queue, unix-style pipe, or sockets. Pipes and sockets are effectively queues managed by the kernel as part of the abstraction. By exposing the fill-level, size, and role of the application (producer or consumer), the scheduler can determine the relative rate of progress of the application by monitoring the fill-level.

- Server

Proportion Specified	Progress Metric	Period Specified	Period Unspecified
Yes	N/A	Real-time	Aperiodic real-time
No	Yes	Real-rate	
	No	Miscellaneous	

Figure 2: Taxonomy of Thread-types for Controller

Servers are essentially the consumer of a bounded buffer, where the producer may or may not be on the same machine.

- Interactive

  Interactive jobs are servers that listen to *ttys* instead of sockets. Since interactive jobs have specific requirements (periods relative to human perception), the scheduler only needs to know that the job is interactive and the *ttys* in which it is interested.

- I/O intensive

  Applications that process large data sets can be considered consumers of data that is produced by the I/O subsystem. As such, they need to be given sufficient CPU to keep the disks busy. Using informed prefetching interfaces such as TIP[16] or Dynamic Sets[19], or delayed write-back buffers for writes, allows the system to monitor the rate of progress of the I/O subsystem as a producer/consumer for a particular job.

- Other

  Some applications are sufficiently unstructured that no suitable symbiotic interface exists, or may be legacy code that predates the interface and cannot be recompiled. In such cases where our scheduler cannot monitor progress, it uses a simple heuristic policy to assign proportion and period based on whether or not the application uses the allocation it is given.

When an application initializes a symbiotic interface (such as by submitting hints, opening a file, or opening a shared queue), the interface creates a linkage to the kernel using a *meta-interface* system call that registers the queue (or socket, etc.) and the application's use of that queue (producer or consumer). We have implemented a shared-queue library that performs this linkage automatically, and have extended the in-kernel pipe and socket implementation to provide this linkage.

3.3 Adaptive Controller

Given the dispatcher and monitoring components, the job of the scheduler is to assign proportion and period to ensure that applications make reasonable progress. Figure 2 presents the four cases considered by the controller: real-time, aperiodic real-time, real-rate, and miscellaneous threads. Real-time threads specify both proportion and period, aperiodic real-time threads specify proportion only, real-rate do not specify proportion or period but supply a metric of progress, and miscellaneous threads provide no information at all.

- Real-time threads

  Reservation-based scheduling using proportion and period was developed in the context of real-time applications [14], applications that have known proportion and period requirements. To best serve these applications, the controller sets the thread proportion and period to the specified amount and does not modify them in practice. Such a specification (if accepted by the system) is essentially a reservation of resources for the application. Should, however, the system be placed under substantial overload, the controller may raise a quality exception and initiate a renegotiation of the resource reservation.

- Aperiodic real-time threads[2]

  For tasks that have known proportion but are not periodic or have unknown period, the controller must assign a period. With reserva-

tions, the period specifies a deadline by which the scheduler must provide the allocation, and hence is more of a jitter-bound than an operating frequency. Too large a period may introduce unacceptable jitter, whereas too small a period may introduce overhead since dispatching happens more often. Without a progress metric with which to assess the application's needs, our prototype uses a default value of 30 milliseconds. This provides reasonable jitter bounds for interactive applications while limiting overhead to acceptable levels.

- Real-rate threads

We call threads that have a visible metric of progress but are without a known proportion or period *real-rate* since they do not have hard deadlines but do have throughput requirements. Examples of real-rate threads are multimedia pipelines, isochronous device drivers, and servers. During each controller interval, the controller samples the progress of each thread to determine the *pressure* exerted on the thread. Pressure is a number between -1/2 and 1/2; negative values indicate too much progress is being made and the allocation should be reduced, 0 indicates ideal allocation, and positive values indicate the thread is falling behind and needs more CPU. The magnitude of the pressure is relative to how far behind or ahead the thread is running.

Figure 3 contains the formula used by the controller to calculate the total pressure on a thread from its progress metrics, or input/output queues. For shared queues, $F_{t,i}$ is calculated by dividing the current fill-level by the size of the queue and subtracting 1/2. We use 1/2 ($F_{t,i} = 0$) as the optimal fill level since it leaves maximal room to handle bursts by both the producer and consumer. $R_{t,i}$ is used to flip the sign on the queue, since a full queue means the consumer should speed up (positive pressure) while the producer should slow down (negative pressure).

2. To be honest, we are unaware of any applications that fall into this category. We have included it in this discussion for completeness.

$$Q_t = G\left(\sum_i R_{t,i} F_{t,i}\right)$$

$$R_{t,i} = \begin{cases} -1 & \text{If t is a producer of i} \\ 1 & \text{If t is a consumer of i} \end{cases}$$

Q_t, the progress pressure, is a measure of the relative progress of thread t using its progress metric(s). $F_{t,i}$ is a value between -1/2 and 1/2, derived from the progress metric i (e.g. buffer fill level), $R_{t,i}$ flips the sign of $F_{t,i}$ for producers. G calculates a PID control function of the queue pressures.

Figure 3: Progress Pressure Equation

The individual progress pressures are then summed and passed to a proportional-integral-derivative (PID) control to calculate a cumulative pressure, Q_t. A PID controller combines the magnitude of the summed pressures (P) with the integral (I) and with the first-derivative (D) of the function described by the summed progress pressures over time. PID control is a commonly applied technique for building controllers to provide error reduction together with acceptable stability and damping [5].

For aperiodic real-rate threads, the controller must also determine the period. Currently, we use a simple heuristic which increases the period to reduce quantization error when the proportion is small, since the dispatcher can only allocate multiples of the dispatch interval. The controller decreases the period to reduce jitter, which we detect via large oscillations relative to the buffer size. The controller determines the magnitude of oscillation by monitoring the amount of change in fill-level over the course of a period, averaged over several periods. Although this heuristic appears to work well for our video pipeline application, we do not have significant experience with its applicability to other domains.

- Miscellaneous threads

The controller uses a heuristic for threads that do not fall into the previous categories. For proportion, the controller approximates the

$$P_t' = \begin{cases} kQ_t & P_t \text{ on target} \\ -C & P_t \text{ too generous} \end{cases}$$

P_t' is the new allocation for thread t calculated from the progress pressure Q_t and the previous allocation P_t. Normally, the controller multiplies the progress pressure by a constant scaling factor to determine the new desired allocation. If the previous allocation overestimated the application's needs, the controller reduces the allocation by a constant factor.

Figure 4: Proportion Estimation Equation

thread's progress with a positive constant. In this way there is constant pressure to allocate more CPU to a miscellaneous thread, until it is either satisfied or the CPU becomes oversubscribed. For period, the controller uses a default period of 30 milliseconds.

Estimating Proportion

After calculating the queue pressure for a thread, the controller must then calculate the new allocation for the thread. Figure 4 presents the equation used by the controller to estimate proportion. In normal circumstances, we multiply the queue pressure by a constant scaling factor to determine the desired allocation. However, increasing the allocation may not improve the thread's progress, as might happen for example if another resource (such as a disk-as-producer) is the bottleneck for this application.

To reclaim the unused allocation, the controller compares the CPU used by a thread with the amount allocated to it.[3] If the difference is larger than a threshold, the controller assumes the pressure is overestimating the actual need and the allocation should be reduced.

3. We assume that the RBS is giving threads as much CPU as the controller allocated, since we reserve some spare capacity. If the RBS is missing deadlines, it notifies the controller which can increase the amount of spare capacity by reducing the admission threshold.

Responding to Overload

When the sum of the desired thresholds is greater than the amount of available CPU, the controller must somehow reduce the allocations to the threads. This increase can result either from the entrance of a new real-time thread, or from the controller's periodic estimation of real-rate or miscellaneous threads' needs. In the former case, the controller performs admission control by rejecting new real-time jobs which request more CPU than is currently available. We chose this approach for simplicity, we hope to extend it to support a form of quality negotiation such as that used in BBN's Quality Objects [22].

In the latter case, the controller *squishes* current job allocations to free capacity for the new request. After the new allocations have been calculated, the controller sums them and compares them to an overload threshold. If the allocations oversubscribe the CPU, it squishes each miscellaneous or real-rate job's proposed allocation by an amount proportional to the allocation. In the absence of other information (such as progress metrics), this policy results in equal allocation of the CPU to all competing jobs over time.

We have extended this simple fair-share policy by associating an importance with each thread. The result is a weighted fair-share, where the importance is the weighting factor. Our use of importance is different than the concept of priority, since a more-important job cannot starve a less important job. Instead, importance determines the likelihood that a thread will get its desired allocation. For two jobs that both desire more than the available CPU, the more important job will end up with the higher percentage.

Note that this squishing solves the same problem addressed by TCP's exponential backoff [9]. Unlike TCP, our controller is centralized and can easily detect overload, allowing us to provide proportional sharing while enforcing compliance.

Implementation

We have implemented this controller using the SWiFT software feedback toolkit [6]. SWiFT embodies an approach to building adaptive system software that is based on control theory. With SWiFT, the controller is a circuit that calculates a function based on its inputs (in this case the

progress monitors and importance parameters), and uses the function's output for actuation.

For reasons of rapid prototyping, our controller is implemented as a user-level program. This has clear implications on overhead, which limits the controller's frequency of execution, which in turn limits its responsiveness. We have plans to move the controller into the Linux kernel in order to reduce this overhead. Nonetheless, our experiments discussed below show the overhead to be reasonable for a prototype system for most common jobs.

In our prototype, jobs must either explicitly register themselves in order to be scheduled by our RBS scheduler (as opposed to the default Linux scheduler) or be descended from such a job. In the future, we hope to schedule all jobs using our scheduler. Currently we limit it to specific jobs such as real-time applications, the controller process, and the X server.

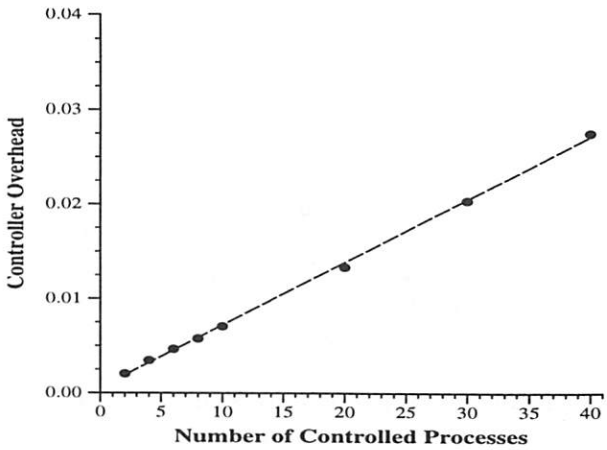
4 Discussion

The following sections discuss various aspects of our solution in more detail. Section 4.1 characterizes our prototype's performance. Section 4.2 examines the responsiveness of our controller to variable-rate real-rate applications, with and without competing load. Section 4.3 discusses ways to improve the accuracy and responsiveness of the system, while Section 4.4 justifies our claims about the benefits of our approach. The experiments were run on a 400 Mhz Pentium 2 with 128MB of memory, running our modified version of Linux 2.0.35.

In all the experiments, we disabled the period estimation aspect of the controller. Period adjustment and buffer size are inter-related, since both are used to reduce jitter and both affect the likelihood of completely filling or emptying the buffer. A proper discussion of the interactions between period adjustment and buffer size adjustment are unfortunately beyond the scope of this paper.

4.1 Characterization

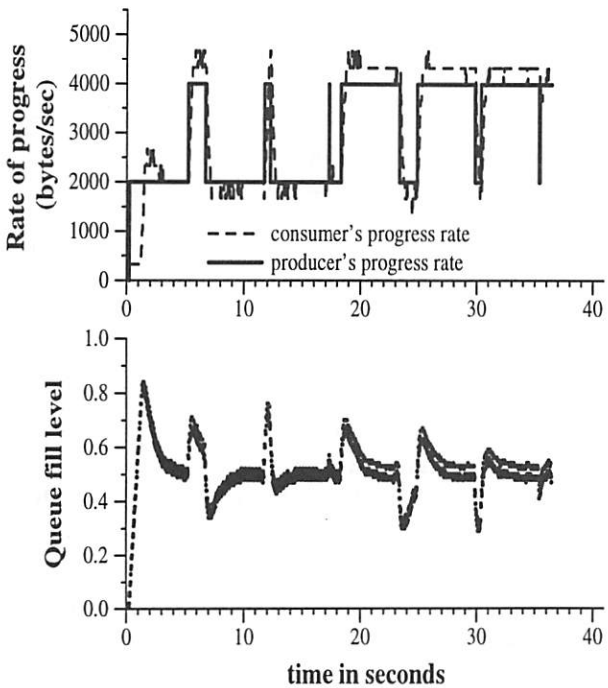
To better understand the characteristics of our system, we discuss its overhead and responsiveness; presenting an analysis of its stability is beyond the scope of this paper. At the lowest level, the overhead of dispatch depends on the execution time of two routines in the Linux scheduler, `schedule()` and `do_timers()`. `Schedule()` is called on dispatch and runs in time linear with the number of threads on the run queue, and in the worst case linear with the number of threads in the system. `Do_timers()` is called on timer interrupts, checks for expired timers, and moves threads waiting on expired timers to the run-queue; preempting the current thread if the woken thread is under our control and has higher goodness. We keep a list of timers used by RBS threads, sorted by time of expiry, and cache the next expiration time to avoid doing any work unless at least one timer has expired. As a result, this routine typically runs in constant time, but in the worst case runs linearly with the number of threads under our control.



This figure shows the overhead of our user-level controller. Our experimental results are linear, y = .00066x + .00057, with a coefficient of determination of .999. The y-axis is the amount of CPU consumed by the controller, where 1 corresponds to 100% utilization. For 40 jobs (x = 40), the overhead is 2.7% of CPU capacity.

Figure 5: Overhead of Controller

To assess the overhead of our use of feedback, we measured the overhead of our user-level controller. Figure 5 depicts the controller's overhead in terms of additional CPU utilization, where the first process is the controller itself, running with a 10 msec period and the additional processes are dummy processes that consume no CPU but are scheduled, monitored, and controlled. At each controller period, the controller must read the progress metrics from the kernel, calculate new allocations, and send the new values to the in-kernel RBS. As a result, the overhead of the controller grows linearly



This figure shows the response of the controller to a variable-rate real-rate job. The producer runs at a predetermined variable rate, the controller determines the consumer's allocation so that its progress matches that of the producer. The top graph shows the progress rates of the producer and consumer, the bottom graph shows the corresponding queue fill-level.

Figure 6: Controller Responsiveness

with the number of threads it controls. The slope of this line is small, .064% of the CPU per process, even though our prototype is not an optimal implementation.

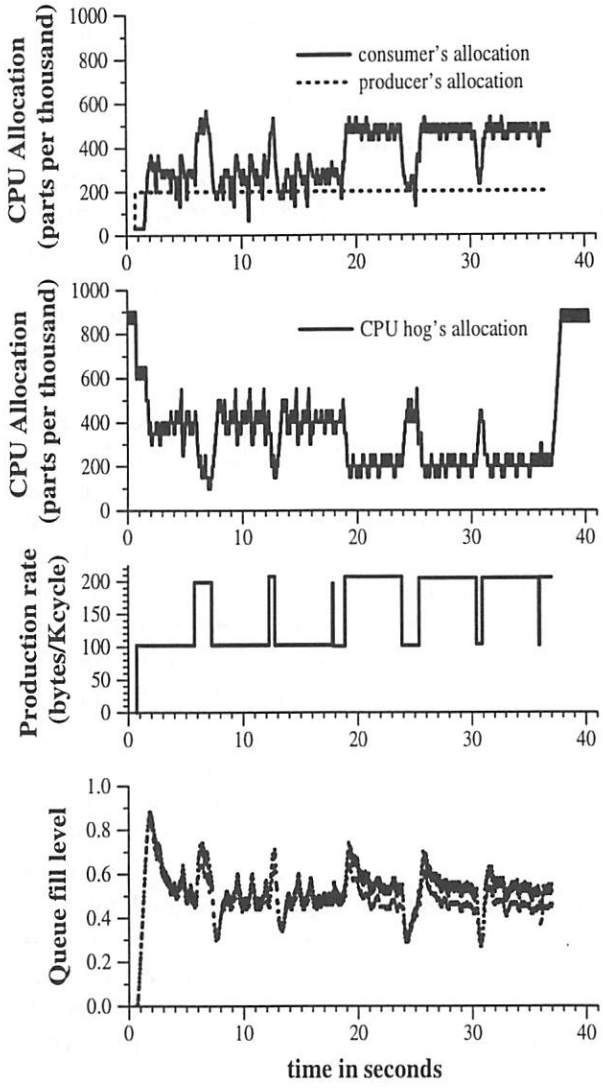
Overhead that is linear with the number of threads under control is a necessary evil for feedback-controlled systems. The benefit is that the system dynamically and rapidly detects changes in the threads' resource requirements, which results in very efficient resource utilization. Fortunately, this recalculation need only happen at the rate at which a process's needs change, and not as often as thread dispatch. The controller is *sampling* the resource needs of the threads, and need only sample twice as fast as the highest rate of change. Using a suitable low-pass filter, we can schedule jobs with reasonable responsiveness and low overhead while keeping the sampling rate reasonably high (100 Hz in our prototype). For example, we currently schedule both the controller and the X server, and see no noticeable delays in interactive response time even when the CPU is fully utilized.

4.2 Controller Responsiveness

To characterize the responsiveness of our system, we wrote a program that simulates a pulse function for our controller. The program is a simple pipeline of a producer and consumer connected by a bounded buffer. Both the producer and consumer loop for some number of cycles before they enqueue or dequeue a block of data. We fix the allocation (cycles/sec) given to the producer by specifying a reservation for it, and control the rate at which it produces data (bytes/cycle). For the consumer, we fix the rate of consumption, but let the controller determine the allocation. Ideally, the producer's rate of progress in bytes/sec should match the consumer's. By manipulating the producer's production rate we can determine the responsiveness of the controller as it adjusts the consumer's allocation to achieve the same rate of progress.

Figure 6 shows the results of running this program on an otherwise idle system. The producer generated rising pulses of various widths, doubling its rate of production in bytes/cycle for a period of time before falling back to the original rate. To maintain the queue at half-full, the controller must double the allocation to the consumer since the producer has specified its proportion and period, the controller does not affect its allocation. After running for three rising pulses, the producer keeps its default rate high and generates three falling pulses.

Figure 6 contains two graphs, the rates of progress of the producer and consumer calculated by multiplying the measured allocation in cycles/sec by the controlled rate of production or consumption in bytes/cycle, and the measured queue fill level in the bounded buffer between the threads. As shown in the graphs, the controller responds to the change in the producer's rate by rapidly increasing the allocation to the consumer, even though its knowledge of either the producer or consumer is limited to their use of the queue. In addition, the shape of the fill level curve and the consumer's allocation match our expectations: the allocation roughly follows the square wave set by



This figure shows the same pipeline run concurrently with a CPU hog. Since the total desired allocation exceeds the capacity of the CPU, the controller must squish the load and consumer threads. It cannot squish the producer since the producer has specified a fixed reservation. Note that the Y-axis in the top graph in Figure 6 has different units than those used here.

Figure 7: Controller Response Under Load

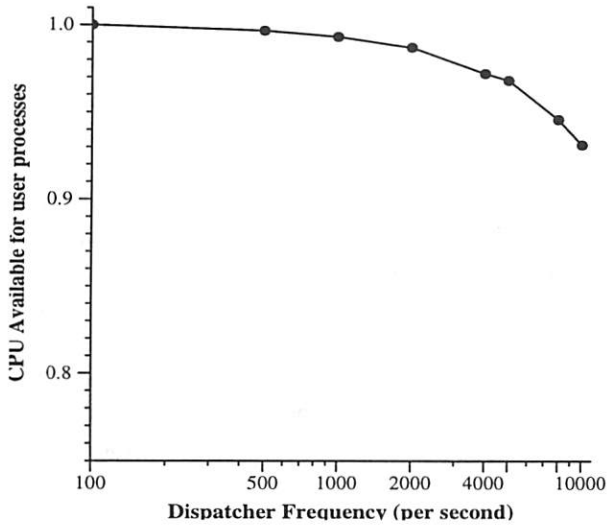
the production rate, and the fill level changes more drastically the farther it is from 1/2. The effect on fill level from pulses with smaller width is smaller, because the producer has less time to produce more data. From our data, it takes the controller roughly 1/3 of a second to respond to the doubling in production rate.

Figure 7 shows the same experiment run with competing load. For simplicity, the load corresponded to a miscellaneous job (no progress-metric) that tries to consume as much CPU as it can. The effect of the competing load is that the total desired allocation of the producer, consumer, and load threads is greater than the amount available, and hence the controller must squish allocations. The top two graph shows the resulting allocations to these three jobs (separated for clarity). The producer's allocation is fixed because it has specified a reservation. The load's allocation is initially high, but effectively loses allocation to the consumer since its pressure Q_t is constant over time while the consumer's grows as it falls further behind the producer. If it were the case that there was not sufficient CPU to satisfy all the jobs, the queue would eventually become full and trigger a quality exception, allowing the application to adapt by lowering its resource requirements.

One interesting result is the high frequency oscillation in allocation between the load and the consumer. This oscillation results from changes in the relative pressures from the hog and the consumer. When the consumer matches the fixed rate of the producer, its Q_t is low while the hog's Q_t remains constant. As it falls behind, its Q_t grows until it exceeds that of the hog, and it gets more allocation. This behavior matches our expectation for real-rate jobs which must track some real-world rate, such as the rate of requests arriving at a web server. We believe that in the future most jobs will have progress metrics, and the use of a constant pressure, and hence the occurrence of such oscillation, will be infrequent.

4.3 Improving the Controller's Behavior

We have several ideas for increasing the accuracy of the allocation. First, we plan to lower the overhead of the controller in order to run it at a higher frequency. Calculating the exponential and linear curves more frequently causes the allocation to change faster, and results in a more responsive system without affecting its stability. In some sense a priority-based scheme is perfectly responsive, but is also inherently unstable.



This figure shows the overhead of dispatch vs. the size of the dispatch interval. The graph shows the amount of CPU available to processes, the area above the curve is the dispatch overhead. There is a knee around 4000Hz. At this point the overhead is around 2.7%.

Figure 8: Dispatch Overhead vs. Frequency

Second, our controller currently suffers quantization errors because the minimum allocation is 1 msec. We are currently exploring possible solutions to this. One possibility is a more efficient dispatch algorithm that can be run at a higher rate. Another possibility is to provide better accounting, e.g., microsecond granularity, while keeping the dispatch interval at 1 msec. This reduces our ability to guarantee proportion since we cannot prevent a job from running for its full time-slice. However, our controller could preempt earlier if the thread makes a system call or an interrupt occurs. In addition, our use of feedback could account for instantaneous discrepancies by smoothing allocation over time.

To determine the overhead of smaller dispatch quanta, we measured the overhead of running Linux with various time-slice lengths. We measured the amount of CPU available to applications by running a program that attempts to use as much CPU as it can. Figure 8 shows the results of this experiment. The number plotted is the amount of CPU the program was able to grab, normalized to the amount it can grab on a kernel with a time-slice of 10msec. The graph shows the results of the higher overhead for smaller quanta, with a knee around 4000 Hz (250 µsec). We conjecture that we could run with a dispatch interval in the range of 50 µsec on faster CPUs with a small effort to optimize our code.

4.4 Benefits of Real-Rate Scheduling

The benefit of scheduling based on progress is that allocation is automatically scaled as the application's requirements change. In our system, the amount of CPU given to a thread grows in relation to its progress pressure and importance. For example, we have a multimedia pipeline of processes that communicate with a shared queue. Our controller automatically identifies that one stage of the pipeline has vastly different CPU requirements than the others (the video decoder), even though all the processes have the same priority. This results in a more predictable system since its correctness does not rely on applications to be well-behaved. In other words, when a real time job spins instead of blocking, the system will not livelock.

Another benefit is that starvation, and thus priority inversion, cannot occur. Dependent processes (connected (in)directly by progess metrics) cannot starve each other since eventually one will block when its fill-level reaches full or empty. Further, dependent processes can dynamically achieve stable configurations of CPU sharing that fair-share, weighted fair-share, or priorities cannot. For independent non real-rate threads, we prevent starvation through our fair-share or weighted fair-share policies. In particular, one process cannot keep the CPU from another process indefinitely simply because it is more important.

A third benefit of our approach is that it automatically provides both "best-effort" and "real-time" scheduling, in addition to the real-rate scheduling that motivates the work. However, we believe the real-rate category to be the most important of the three in the future, as people use computers to interact with each other and with the real world.

4.5 Effect on Miscellaneous Applications

Although the importance of real-rate applications such as speech recognition, multimedia, and Web servers will grow to dominance in the future, many PCs still run a mix of more traditional applications that have no rate requirements and for which priorities have sufficed. For these applica-

tions, our approach can potentially reduce performance (modulo responsiveness). However, these applications can still suffer from priority inversion and starvation, even if they do not benefit from predictable scheduling and fine-grain control. We suggest the right solution for these applications is to add a pseudo-progress metric which maps their notion of progress into our queue-based meta-interface. For example, a pure computation (finding digits of π or cracking passwords) could use a metric such as the number of keys it has attempted. This could be done transparently by augmenting the in-kernel resources such as ttys or sockets to expose fill-levels to the scheduler. Although we might be able to improve the performance of our scheduler for miscellaneous jobs, we believe jobs with no time or rate requirements will be uncommon in the future and thus such an effort is likely to have small returns relative to those gained by converting the jobs to be real-rate.

5 Related Work

There exists a large body of work which has attempted to provide real-time scheduling support in operating systems, Jones et al. [11] provide a nice summary. Linux, Solaris, and NT provide "real-time" priorities, which are fixed priorities that are higher in priority than regular priorities. More relevant to this work are efforts to schedule based on proportion and/or period [11][15][20][21]. To date, all such approaches require human experts to supply accurate specifications of proportion and/or period, and focus on how to satisfy these specifications in the best way. None of them try to infer the correct proportion, or adapt dynamically to changing resource needs of the applications.

In addition, several systems use hybrid approaches to merge the benefits of reservation and priority scheduling. Typically these approaches use a heuristic that gives a static[4][8] or biased [7] partition of the CPU to either real-time jobs or non-real-time jobs. A new approach is taken by the BERT and SMaRT schedulers, which dynamically balances between the needs of both kinds of jobs. The SMaRT scheduler lets users assign priority to either conventional or real-time threads, but gives weight to non-real-time threads within the same equivalence class [15]. Although we implicitly give precedence to real-time tasks (those that specify both proportion and period), we expect most jobs to fall into the real-rate category. This includes all of what most people consider "soft-real-time" applications such as multimedia.

The BERT scheduler handles both real-time and non-real-time tasks using the same scheduling mechanism. Jobs submit units of work to be scheduled, and the scheduler creates deadlines for the work based on previous measures of the work's time to completion. BERT automatically assesses whether a given job will meet its deadline, and if not can either steal cycles from a lower priority job or can cancel the job[1]. BERT is similar in philosophy to our approach since it uses feedback of past execution times in its scheduling, but it does not use or measure application progress and as such is subject to the same problems as traditional schedulers.

Our solution is similar to Rate-based scheduling proposed by Jeffay and Bennett [10], in that resources are allocated based on rate specifications of x units of execution every y time units. However, their units are events which are converted to CPU cycles using a worst-case estimate of event processing time. Applications must specify x, y, and the worst-case estimation, and an upper-bound on response time. In addition, these values are constant for the duration of the application Their system also uses pipelines of processes so that dependent stages do not need to specify their rate, merely their event processing time. In contrast, our system provides dynamic estimation and adjustment of rate parameters, and only requires that the process metric be specified.

In short, to the best of our knowledge we are the first to attempt to schedule using feedback of the application's rate of progress with respect to its inputs and/or outputs. The power of this approach lets us provide a single uniform scheduling mechanism that works well for all classes of applications, including real-time, real-rate, and conventional.

6 Conclusion

Real-rate applications that must match their throughput to some external rate, such as web servers or multimedia pipelines, and real-time applications are poorly served by today's general purpose operating systems. One reason is that priority-

based scheduling, widely used in existing operating systems, lacks sufficient control to accommodate the dynamically changing needs of these applications. In addition, priority-based scheduling is subject to failure modes such as starvation and priority inversion that reduce the robustness of the system.

In this paper we have described a new approach to scheduling that assigns proportion based on measured rate of progress. Our system utilizes progress monitors such as the fill-level in a bounded buffer, a feedback-based controller that dynamically adjusts the CPU allocation and period of threads in the system, and an underlying proportional reservation-based scheduler. As a result, our system dynamically adapts allocation to meet current resource needs of applications, without requiring input from human experts.

7 Bibliography

[1] A. Bavier, L. Peterson, and D. Moseberger. BERT: A scheduler for best effort and realtime tasks. Technical Report TR-587-98, Princeton University, August 1998.

[2] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals,* pages 47-50. Addison Wesley, second edition, 1998. Translated from the German edition Linux-Kernel-Programmierung published by Addison-Wesley GmbH.

[3] F. J. Corbato, M. Merwin-Daggett, and R.C. Daley. An Experimental Time-Sharing System. *Proceedings of the AFIPS Fall Joint Computer Conference.* 1962. As cited in *Operating System Concepts,* page 153. A. Silberschatz and P. B. Galvin. Addison-Wesley, 5th edition.

[4] B. Ford and S. Susarla. CPU inheritance scheduling. In *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation.* Seattle, WA. October 1996.

[5] G. F. Franklin, J. D. Powell, and A. Emami-Naeini. *Feedback Control of Dynamic Systems,* page 185. Addison-Wesley, third edition, 1994. Reprinted with corrections June, 1995.

[6] A. Goel, D. Steere, C. Pu, and J. Walpole. SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit. Technical Report CSE-98-009, Department of Computer Science and Engineering, Oregon Graduate Institute. June 1998.

[7] R. Govindan and D. P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proceedings of the 13th ACM Symposium on Operating System Principles,* pages 68-80, October 1991.

[8] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation.* Seattle, WA. October 1996.

[9] V. Jacobson. Congestion avoidance and control. In *Proceedings of the SIGCOMM '88 Conference on Communications Architectures and Protocols,* 1988.

[10] K. Jeffay, and David Bennett. A rate-based execution abstraction for multimedia computing. In *Proceedings of the Fifth International Workshop on Network and Operating System Support for Digital Audio and Video,* Durham, NH, April 1995. Published in *Lecture Notes in Computer Science,* T.D.C. Little and R. Gusella, editors. Volume 1018, pages 64-75. Springer-Verlag, Heidelberg, Germany, 1995.

[11] M. B. Jones, D. Rosu, and M-C. Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating System Principles,* pages 198–211, October 1997.

[12] Mike B. Jones. What really happened on Mars. Email, available on the Web at http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html.

[13] B. W. Lampson and D. D. Redell. Experience with processes and monitors in mesa. *Communications of the ACM,* 23(2):105–117, 1980. Also appeared in Proceedings of the 7th ACM Symposium on Operating System Principles, Pacific Grove, CA, 1979.

[14] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM, 20(1):46-61, January 1973.

[15] J. Nieh and M. S. Lam. The design, implementation, and evaluation of SMaRT: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating System Principles,* pages 184–197, October 1997.

[16] R. H. Patterson, G. A. Gibson, E. Ginting, D.

Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[17] Glenn Reeves. Re: What really happened on mars. Email, available on the Web at http://research.microsoft.com/~mbj/Mars_Pathfinder/Authoritative_Account.html.

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, September 1990.

[19] D. C. Steere. Exploiting the Non-determinism and Asynchrony of Set Iterators to Reduce Aggregate File I/O Latency. *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 252-263, October 1997.

[20] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the Duality between resource reservation and proportional share resource allocation. In *Multimedia Computing and Networking 1997*. SPIE Proceedings Series, Volume 3020. San Jose, CA, February 1997, pages 207-214.

[21] C. A. Waldspurger, and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating System Design and Implementation*. November 1994, pages 1-11.

[22] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for corba objects. *Theory and Practice of Object Systems*, April 1997. http://www.dist-systems.bbn.com/papers/TAPOS.

A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98

Erik Cota-Robles
James P. Held
Intel Architecture Labs

Abstract

Windows† 98 and NT† share a common driver model known as WDM (Windows Driver Model) and carefully designed drivers can be binary portable. We compare the performance of Windows 98 and Windows NT 4.0 under load from office, multimedia and engineering applications on a personal computer (PC) of modest power that is free of legacy hardware. We report our observations using a complementary pair of system performance measures, *interrupt* and *thread latency*, that capture the ability of the OS to support multimedia and real-time workloads in a way that traditional throughput-based performance measures miss. We use the measured latency distributions to evaluate the quality of service that a WDM driver can expect to receive on both OSs, irrespective of whether the driver uses thread-based or interrupt-based processing. We conclude that for real-time applications a driver on Windows NT 4.0 that uses high, real-time priority threads receives an order of magnitude better service than a similar WDM driver on Windows 98 that uses Deferred Procedure Calls, a form of interrupt processing. With the increase in multimedia and other real-time processing on PCs the interrupt and thread latency metrics have become as important as the throughput metrics traditionally used to measure performance.

1. Introduction

Real-time computations in multimedia applications and device drivers are typically performed in response to interrupts or the completion of previous computations that were themselves performed in response to interrupts. Under the Windows Driver Model (WDM) [1][21] such computations are typically implemented as either Deferred Procedure Calls (DPCs)[1][21], a form of interrupt processing, or in kernel mode threads. The ability of applications and drivers to complete their computations before their respective deadlines is thus a function of the expected worst-case delay between the hardware interrupt and the start of the computation.

† Third-party brands and names are the property of their respective owners.

These delays, or *latencies*, are highly sensitive to the amount of OS overhead that is incurred to service any other applications that may be executing concurrently on the system. Traditional real-time systems cope with this problem by strictly limiting the amount of concurrent non-real-time computation and by using a real-time OS with tightly bounded service times. This minimizes the amount of overhead penalty to which any one computation is subjected. On personal computer and workstation platforms the execution environment is highly dynamic and may include a wide variety of concurrently executing applications whose nature can only be estimated in advance. It is therefore not practicable to either limit application concurrency or to use a real-time OS.

Application (low latency streaming**)	Buffer size in ms. (t) (typ. range)	Number of buffers (n) (typ. range)	Latency Tolerance (n-1)*t ms.
ADSL	2 to 4	2 to 6	4 to 10
Modem	4 to 16	2 to 6	12 to 20
RT audio**	8 to 24	2 to 8[1]	20 to 60[1]
RT video**	33 to 50	2 to 3	33 to 100

Table 1: Range of Latency Tolerances for Several Multimedia and Signal Processing Applications, tolerance range roughly $(n_{max}\text{-}1)*t_{min}$ to $(n_{min}\text{-}1)*t_{max}$ ms.

Applications and drivers vary widely in their tolerance for missed deadlines, and it is often the case that two drivers with similar throughput requirements must use very different kernel services (e.g., DPCs and kernel mode threads). Before an application or driver misses a deadline all buffered data must be consumed. If an application has n buffers each of length t, then we say that its *latency tolerance* is $(n\text{-}1) * t$. Table 1 gives latency tolerance data for several applications [5][11]. It is interesting to note that the two most processor-

[1] 8 is the maximum number of buffers used by Microsoft's KMixer and is on the high side. 4 buffers, which yields a latency tolerance of 20 to 40 milliseconds, would be more realisitic for low latency audio.

intensive applications, ADSL and video at 20 to 30 fps, are at opposite ends of the latency tolerance spectrum. Traditional methodologies for system performance measurement focus on throughput and average case behavior and thus do not adequately capture the ability of a computing system to perform real-time processing.

In this paper we propose a new methodology for OS performance analysis that captures the ability of a non-real-time computing system to meet the latency tolerances of multimedia applications and low latency drivers for host-based signal processing. The methodology is based on a complementary pair of microbenchmark measures of system performance, *interrupt latency* and *thread latency*, which are defined in section 2.1. Unlike previous microbenchmark methodologies, we make our assessments of OS overhead based on the distribution of individual OS service times, or latencies, on a loaded system. We present extremely low cost, non-invasive techniques for instrumenting the OS in such a way that individual service times can be accurately measured. These techniques do *not* require OS source code, but rely instead on hardware services, in particular the Pentium® and Pentium II processors' time stamp counters[9][10], and can thus be adapted to any OS.

We use these techniques to compare the behavior of the Windows Driver Model (WDM) on Windows 98 and Windows NT under load from a variety of consumer and business applications. We show that for real-time applications a driver on Windows NT 4.0 that uses either Deferred Procedure Calls (DPCs), a form of interrupt processing, or real-time priority kernel mode threads will receive service at least one order of magnitude better than that received by an identical WDM driver on Windows 98. In fact, a driver on Windows NT 4.0 that uses high, real-time priority threads will receive service one order of magnitude better than a WDM driver on Windows 98 which uses DPCs. In contrast, traditional throughput metrics predict a WDM driver will have essentially identical performance irrespective of OS or mode of processing.

The remainder of this section provides background on prior work on OS benchmarking and the performance analysis of real-time systems. Section 2 presents our methodology for OS performance analysis, including definitions of the various latencies in which we are interested and a description of our tools and measurement procedures. Section 3 presents our application stress loads and test system configuration. Section 4 presents and discusses our results. In section 5 we explore the implications of our results for hard real-time drivers, such as soft modems, on Windows 98 and Windows NT. Section 6 concludes.

1.1 Macrobenchmarks

Traditional methodologies for system performance measurement focus on overall throughput. Batch macrobenchmarks, whether compute-oriented such as SpecInt† or application-oriented such as Winstone† [22], drive the system as quickly as possible and produce one or a few numbers representing the time that the benchmark took to execute. While appropriate for batch and time-shared applications this type of benchmarking totally ignores significant aspects of system performance. As Endo, et. al. note [7], throughput metrics do not adequately characterize the ability of a computing system to provide timely response to user inputs and thus batch benchmarks do not provide the information necessary to evaluate a system's interactive performance.

Current macrobenchmarks essentially ignore latency with the result that the resulting throughput analysis is an unreliable indication of multimedia or real-time performance. For multimedia and other real-time applications, increased throughput is only one of several prerequisites for increased application performance. Because macrobenchmarks do not provide any information about the distribution of OS overhead, they do not provide sufficient information to judge a computing system's ability to support real-time applications such as low latency audio and video.

1.2 Microbenchmarks

Microbenchmarks, in contrast, measure the cost of low-level primitive OS services, such as thread context switch time, by measuring the average cost over thousands of invocations of the OS service on an otherwise unloaded system. The principal motivations for these choices appear to have been the limited resolution of traditional hardware timers and a desire to distinguish OS overhead from hardware overhead by using warm caches, etc. The result, as Bershad *et. al.* note [2], is that microbenchmarks have not been very useful in assessing the OS and hardware overhead that an application or driver will actually receive in practice.

Most previous efforts to quantify the performance of personal computer and desktop workstation OSs have focused on average case values using measurements conducted on otherwise unloaded systems. Ousterhout evaluates OS performance using a collection of microbenchmarks, including time to enter/exit the kernel, process context-switch time, and several file I/O benchmarks [19]. McVoy and Staelin extend this work to create a portable suite, *lmbench*, for measuring OS as well as hardware performance primitives [17]. Brown and Seltzer extend *lmbench* to create a more robust,

flexible and accurate suite, *hbench:OS*, which utilizes the performance counters on the Pentium and Pentium Pro processors to instrument the OS [3].

For the purposes of characterizing real-time performance, all of these benchmarks share a common problem in that they measure a subset of the OS overhead that an actual application would experience during normal operation. For example, Brown and Seltzer revise the *lmbench* measurement of context switch time so as to exclude from the measurement any effects from cache conflict overhead due to faulting of the working set of a new process. The motivation given is that by redefining context switch time in this manner the *hbench:OS* can obtain measurements with a standard deviation an order of magnitude less than those produced by *lmbench*. While this accurately characterizes the actual OS cost to save/restore state, one must in addition use another microbenchmark to measure cache performance and then combine the two measurements in an unspecified manner in order to obtain a realisitic projection of actual application performance. Furthermore, none of these OS micro-benchmarks directly addresses response to interrupts, which is of prime importance to low latency drivers and multimedia applications.

In contrast to the OS microbenchmarks discussed above, Endo, et. al., develop microbenchmarks based on simple interactive events such as keystrokes and mouse clicks on personal computers running Windows NT and Windows 95 [7]. They also construct activity-oriented *task* benchmarks designed to model specific user actions when using popular applications such as Microsoft Word. These benchmarks do address response to interrupts and detailed distributions are reported for some of the data. However, the authors' focus is on interactive response times, which for low-level input events, such as mouse and keyboard, is generally regarded as being adequately responsive if the latencies are in the range of 50 to 150 ms. [20] . As we have seen above (Table 1), except for video, this is considerably longer than the latency tolerances of the low latency drivers and multimedia applications that we consider here, which have tolerances between 4 and 40 milliseconds, depending on the specific application.

1.3 Real-Time Systems

Efforts to characterize the behavior of real-time systems, both software and hardware, have focused largely on worst-case behavior and assumed that the overall system load is known in advance. Katcher *et. al.*, for example, decompose OS overhead into the four categories of preemption, exit, non-preemption and timer interrupts [13]. *Preemption* is the time to preempt a thread; *exit*, the time to resume execution of a previously preempted thread; and *non-preemption*, the blocking time due to interrupt handling which does not result in preemption (i.e., the thread is added to the ready queue).

For systems with a fixed priority preemptive scheduler, it is common to use Rate Monotonic Analysis (RMA) to determine whether each of the system's threads can be scheduled so as to complete before its deadline. Traditionally this has been done by ignoring OS overhead [15], but recently techniques have been developed to include worst-case OS behavior into the analysis [14]. While such models are comprehensive and adequate for real-time OSs, they are overly pessimistic for Windows, which has worst case times for system services, such as context switching, that are orders of magnitude longer than average case times.

A further complication is that computationally intensive drivers, such as those for host-based signal processing, perform significant amounts of processing at "high priority" (e.g., in an interrupt service routine (ISR)). As an example, the datapump[2] for a software modem will typically execute periodically with a cycle time of between 4 and 16 milliseconds and take somewhat less than 25% of a cycle (i.e., 1 to 4 milliseconds) on a personal computer with a 300 MHz Pentium II processor. Clearly, multi-millisecond computations in an ISR will impact both interrupt and thread latency; they will also render a traditional worst-case analysis still more pessimistically. In previous work we have shown how RMA can be extended to general-purpose OSs that have highly non-deterministic OS service times in order to obtain reasonable estimates of real-time performance [4]. We will return to this subject in section 5.2.

2. Methodology

We sought a small set of microbenchmarks that would encapsulate the effects of OS overhead from a real-time standpoint but could be manageably incorporated into a performance analysis in order to accurately forecast the real-time performance of Windows applications and drivers. Since our goal was for the benchmarks to be applicable to a variety of real-time applications, we avoided task-oriented benchmarks of the type used by Endo, et. al., [7] in favor of general microbenchmarks.

[2] The datapump is the modem physical interface layer, analogous to the OSI PHY layer for networks.

Because user mode applications can be a noticeable impediment to timely response by the operating system, we measured latency in the presence of the stress from unrelated applications. This approach is valid even for assessing the performance that real-time portions of large multimedia applications will receive with no concurrent applications. Indeed, from the standpoint of low level real-time drivers (e.g., a kernel mode soft modem or low latency soft audio codec) the rest of the application (e.g., the user mode video codecs or the GUI display) is, for all practical purposes, an external application load.

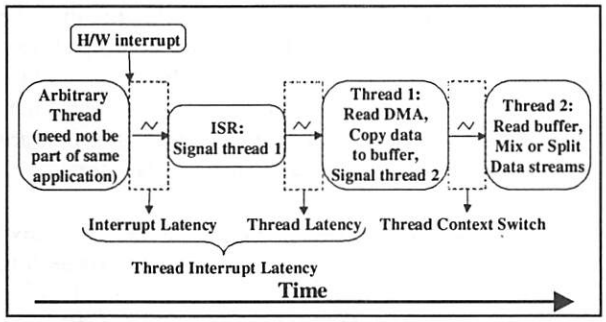


Figure 1: Interrupt Latency, Thread Latency and Thread Context Switch Time

2.1 Latency

Interrupt latency is defined to be the delay from the assertion of the hardware interrupt, *as seen by the processor*, until the first instruction of the software interrupt service routine (ISR) is executed. Thus, it measures the total delay to initial servicing of an interrupt. This encompasses the maximum time during which interrupts are disabled as well as the bus latency necessary to resolve the interrupt, but does not include the bus latency prior to the assertion of the interrupt at the processor. Figure 1 depicts an idealized timeline with interrupt latency, thread latency and thread context switch time marked.

Thread latency is defined to be the delay from the time at which an ISR signals a waiting thread until the time at which the signaled thread executes the first instruction after the wait is satisfied. Thus, it measures the worst-case thread dispatch latency for a thread waiting on an interrupt, measured from the ISR itself to the first instruction executed by the thread after the wait. Thread latency encompasses a variety of thread types and priorities (e.g., kernel mode high real-time priority) and includes the time required to save and restore a thread context and obtain and/or release semaphores. It represents the maximum time during which the operating system disables thread scheduling. An important point to note is that thread latency subsumes thread context-switch time since, in the general case, the proper thread is not executing when an interrupt arrives. We distinguish between thread latency, defined above, and *thread interrupt latency*, defined to be the delay from the assertion of the hardware interrupt until the thread begins execution.

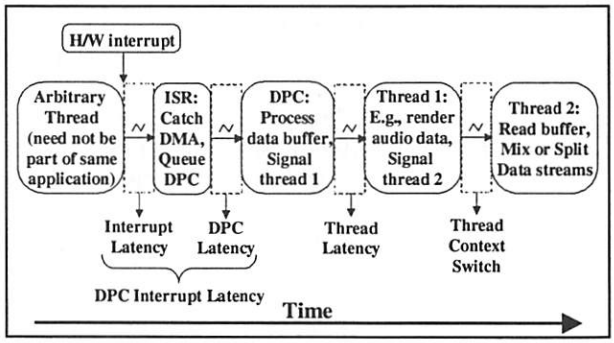


Figure 2: DPC Latency, DPC Interrupt Latency and WDM Thread Latency

In the Windows Driver Model (WDM) interrupts are preemptible and are supposed to be very short [1][21]. WDM makes a Deferred Procedure Call (DPC) available for drivers that require longer processing in "interrupt context". We distinguish between *DPC latency*, which is defined to be the delay from the time at which the software ISR enqueues a DPC until the first instruction of the DPC is executed, and *DPC interrupt latency*, which is defined to be the sum of the interrupt and DPC latencies, as shown in Figure 2. Because ordinary DPCs queue in FIFO order, DPC latency encompasses the time required to enqueue and dequeue a DPC as well as the aggregate time to execute all DPCs in the DPC queue when the DPC was enqueued. Because drivers are not supposed to do substantial processing in a WDM ISR, we will measure WDM thread latencies from DPC to thread, and concentrate on DPC interrupt latency and thread latency.

2.2 Latency Measurement Tools

We developed a number of WDM drivers that measure interrupt and thread latency. The thread latency driver is binary portable between Windows 98 and Windows NT, but the Windows 98 interrupt latency driver uses an interface unique to Windows 9x to install its own timer handler and is thus not portable to NT. The drivers have simple command line control applications and use the Pentium processor's time stamp register (TSR) for timing information. The latencies are returned to the application via WDM I/O Request Packets (IRPs) which the application supplies via a call to the Win32[†] ReadFileEx API. Figure 3 gives an execution timeline and sections 2.2.1 through 2.2.5

present pseudocode for the Windows NT DPC interrupt latency and thread latency tool.

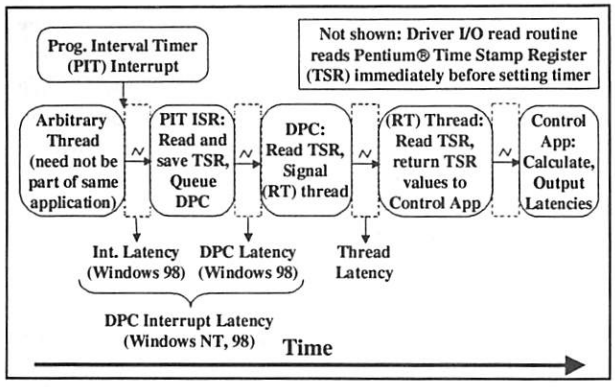


Figure 3: WDM Interrupt, DPC and Thread Latency Measurement Tool

In the case of interrupt latency, because the driver cannot read the time stamp counter at the instant when the hardware interrupt is asserted, the driver I/O read routine reads the time stamp register and sets a timer to expire in a given number of milliseconds. The interrupt latency drivers estimate the time stamp at which the timer expired using the time stamp from the I/O read routine and record this as the estimated time stamp for the hardware interrupt. This approach suffers from limited resolution (basically +/- the cycle time of the Programmable Interval Timer (PIT) timer, whose frequency we have increased to 1 KHz). Because we were mainly interested in characterizing the latency "tail", which on Windows 98 extends past 10 milliseconds, we accepted this imprecision with only minor qualms. To put it another way, we are interested in the frequency of long latency events, so we care about the *magnitude* of long latency events and the *number* of short latency events.

Furthermore, on Windows 98 it is possible, using legacy interfaces, to supply our own timer ISR, whereas on Windows NT this would require source code access. Our NT driver thus records only DPC interrupt latency whereas our Windows 98 driver records interrupt latency, DPC latency, and DPC interrupt latency. This is shown in Figure 3.

As the following pseudocode is highly specific to WDM, a few definitions and clarifications are in order:

- **DPC**: Deferred Procedure Call. In WDM an **ISR** can queue a DPC to a FIFO queue to do time-critical work on its behalf. DPCs execute after all ISRs but before paging and threads. This is similar to the Immediate queue for ISR "Bottom Halves" in Linux†.
- **DriverEntry**: This function is called at driver load time and performs all driver initialization.
- **IRP**: I/O Request Packet. Each user mode call to a Win32 driver interface function (e.g., Read) generates an IRP that is passed to the appropriate driver routine.
- **IRP->AssociatedIrp.SystemBuffer**: This is used to transfer data to/from the user mode application. We abbreviate it as **IRP->ASB** and pretend that it is of type `LARGE_INTEGER`.
- **ISR**: Interrupt Service Routine. In the WDM paradigm, ISRs queue **DPC**s to do work on their behalf.
- **PIT**: Programmable Interval Timer. PC hardware timer. By default on Windows 98 and NT it fires at a frequency of 67 to 100 Hz (10 to 15 ms. period). We reset it to 1 KHz (1 ms. period).
- **Real-time Priority**: WDM has 16 real-time priorities, 16 through 31. 24 is the default.
- **Single shot timer**: An OS timer that fires only once. NT 4.0 added periodic OS timers.
- **Synchronization Event**: An event that auto-clears after a single wait is satisfied. Contrast with a **Notification Event**, which satisfies all outstanding waits, as do Unix† kernel events.

2.2.1 DriverEntry Pseudocode

```
Create a single shot timer gTimer.
Create a Synchronization Event gEvent.
Create a kernel mode thread executing
   LatThreadFunc() (section 2.2.4).
Initialize global variable ghIRP
   shared by thread, driver functions.
Set PIT interrupt interval to 1 ms.
```

2.2.2 Driver I/O read Pseudocode

```
Procedure LatRead(IRP) {
   GetCycleCount(&IRP->ASB[0])
// The PIT ISR will enqueue
// LatDpcRoutine in the DPC queue
   KeSetTimer (gTimer,
               ARBITRARY_DELAY,
               LatDpcRoutine)
}
```

2.2.3 Timer DPC Pseudocode

```
// This is called by the kernel when
// the DPC is dequeued and executed
Procedure LatDpcRoutine(IRP) {
   GetCycleCount(&IRP->ASB[1])
   ghIRP = IRP
   KeSetEvent(gEvent)
}
```

2.2.4 Thread Pseudocode

```
Procedure LatThreadFunc() {
   KeSetPriorityThread(
      KeGetCurrentThread(),24);
   loop (FOREVER) {
      WaitForObject(gEvent,FOREVER)
      GetCycleCount(&ghIRP->ASB[2])
// This completes the read, sending
// the data to the user mode app
      IoCompleteRequest(ghIRP)
      ghIRP = NULL
   } /* loop */
}
```

2.2.5 GetCycleCount code

Because not all versions of the Visual C++[†] inline assembler recognize the Pentium RDTSC instruction, the following function is provided.

```
//  Name: GetCycleCount
//  Purpose: Read the Pentium® cycle
//           (timestamp) counter
//  Context: Called by driver to get
//           current timestamp
//
// Copyright (c) 1995-1998 by Intel
// Corporation. All Rights Reserved.
// This source code is provided "as
// is" and without warranty of any
// kind, express or implied.
// Permission is hereby granted to
// freely use this software for
// research purposes.
//
GetCycleCount(
   LARGE_INTEGER *pliTimeStamp) {
   ULONG Lo;
   LONG Hi;

   _asm {
      _emit 0x0f
      _emit 0x31
      mov Lo, eax
      mov Hi, edx
   } /* _asm */
   pliTimeStamp->LowPart = Lo;
   pliTimeStamp->HighPart = Hi;
   return;
} /* GetCycleCount */
```

2.3 Latency Cause Tool

The latency measurement tools, while eminently useful for assessing the ability of an OS to meet the latency tolerances of real-time applications and drivers, give little insight as to the causes of long latency. We desired to engage OS and possibly driver vendors in an effort to improve real-time performance. A significant impediment is that in a commercial environment we do not have access to source code for either OS or any of the drivers which could be responsible for one or more of the long latencies.

We began by modifying our thread latency tool to hook the Pentium processor Interrupt Descriptor Table (IDT) entry for the Programmable Interval Timer (PIT) interrupt. To do this we patch the PIT timer Interrupt Descriptor Table (Pentium Interrupt Dispatch Table) entry to point to our hook function. The hook function updates a circular buffer with the current instruction pointer, code segment and time stamp and then jumps to the OS PIT ISR. We then modified the thread latency tool to report only latencies in excess of a preset threshold and to dump the contents of the circular buffer when it reported a long latency. Post mortem analysis produces a set of traces of active modules and, if symbol files are available[3], functions. In spite of the lack of source code the module+function traces are often quite revealing. Endo and Seltzer describe a similar technique for recording information on system state during long interactive event latencies as part of a proposed tool suite for Windows NT, but anticipate that OS source code will be needed for causal analysis [8].

3. Test Procedure

3.1 Application Stress Loads

For each application category we estimated how many hours per week constitute heavy use for applications belonging to that category. As explained below, based on the estimates we estimated how many hours of data we needed to collect using each stress application. To maximize our breadth of application coverage and improve reproducibility some stress applications are actually benchmarks. These benchmarks are driven by Microsoft Test (MS-Test) at speeds much faster than possible for a human, enabling us to collect data over a shorter period of time than would otherwise have been the case. We collected data for periods of hours, capturing events that occur at frequencies as low as 1 in 100,000 in statistically significant numbers.

[3] OS symbols are available with a subscription to the Microsoft Developer's Network (MSDN) [21].

3.1.1 Office Applications

To represent the class of office applications we used the Business Winstone 97 benchmark [22], which executes eight business productivity applications spanning three categories of business computing:

- Database: Access† 7.0, Paradox† 7.0
- Publishing: CorelDRAW† 6.0, PageMaker† 6.0, PowerPoint† 7.0
- Word Processing and Spreadsheet: Excel 7.0, Word 7.0, WordPro† 96

Each application is installed via an InstallShield†[4] script, run at full speed through a series of typical user actions and then uninstalled. On initial launch Winstone performs a number of hardware checks (e.g., interrogation of the floppy drive) which cause a marked spike in observed OS latencies. We therefore launched Winstone, then started our latency measurement tools and finally launched the benchmark.

The Business benchmark is driven by MS-Test at speeds in excess of human abilities to type and click a mouse. As Endo *et. al.* observe, this results in an unnaturally time-compressed sequence of user input events that should not occur in normal use, resulting in abnormally large batched requests for OS services [7]. We agree with Endo et. al. that these batched requests may be optimized away by the OS, resulting in a lower overall system load during the benchmark than during equivalent human user activity. Nevertheless, we note that long spurts of system activity will still occur because of, for example, file copying, both explicit and implicit (e.g., "save as"). In our experience this type of extended system activity is much more likely to impact response to interrupts, causing long latencies, than any of the batched requests discussed by Endo, et. al. might cause individually or collectively were they not batched. Since we are only using the Winstone benchmark to impose load, we exploit this time-compression to collect data for a shorter period of time.

Data as to how long a "typical" user would take to execute the Business Winstone 97 benchmark input sequence are unavailable [23], but we can derive a conservative lower bound to the compression ratio under very weak assumptions. To do this we assume that a typing speed of 120 5-character words per minute (or about 1 character every 100 milliseconds) is the upper limit of sustainable human input speed. Based on the default PC clock interrupt rate of 67 to 100 Hz. (see section 2.2) it is clear that Winstone can drive input at least ten times as quickly as a human, even without compensating for the complete absence of "think time" [20] during the benchmark. Thus we estimate that Business Winstone 97 running continuously will produce at least as much system stress in 4 hours as a heavy user will produce in a 40-hour work week.

3.1.2 Workstation applications

To represent the class of workstation applications we used the High-End Winstone 97 automated benchmark [22], which executes six workstation applications spanning three categories of workstation computing:

- Mechanical CAD: AVS† 3.0, Microstation† 95
- Photoediting: Photoshop† 3.0.5, Picture Publisher† 6.0, P-V Wave† 6.0
- S/W Engineering: Visual C++ 4.1 Compiler

As with the Business benchmark, each application is installed via an InstallShield script, run at full speed through a series of typical user actions and then uninstalled. Again, we first launched Winstone, then started our latency measurement tools and finally launched the benchmark.

Workstation applications are inherently more stressful than business applications, and are CPU, disk or network bound (i.e., not waiting on user I/O) more of the time than business applications. We therefore assumed a more conservative 5 to 1 ratio of MS-Test input speed to human input speed. Thus we estimate that 6 hours of continuous testing will produce as much system stress as a heavy user will produce in one work week of 30 hours, assuming engineers spend 2 hours daily using non-engineering applications such as email.

3.1.3 Multimedia Applications

We divided the class of multimedia applications into two subcategories: 3D games and Web browsing with enhanced audio/video. In order to compare apples to apples, we limited ourselves to 3D games that run on Windows 95/98 and Windows NT. Two were selected: Freespace† Descent† and Unreal†. Since game demos are essentially canned sequences of game play, we do not assume any speedup when collecting our 3D game data. We estimate that game enthusiasts play on the order of 2 to 3 hours per day, 4 to 6 days per week, concluding that 12.5 hours of data will capture a week of game play by an enthusiast.

Web browsing is dominated by download times. With a modem on a regular phone line a heavy user is bandwidth limited. By using an Ethernet LAN

[4] InstallShield is a standard Windows application that installs and configures other applications. It is driven by scripts written by the software vendor.

connection, downloading occurs at speeds far in excess of those achievable on a regular phone line. As a result, the system is stressed more than would actually occur during normal usage, and it is not necessary to collect data for as long a period of time as would otherwise be the case. Assuming conservatively a 10 to 1 ratio of 10 MBit Ethernet download speed to regular phone line download speed, we estimate an overall 4 to 1 ratio, given that the user also spends time reading Web pages, listening to audio and video clips, etc. We estimate that a heavy user browses the Web about 3 to 4 hours per day, 7 days per week. We conclude that 8 hours of data while browsing with an Ethernet network connection should capture about a week of Web browsing over a regular phone line by a heavy user.

We split our Web testing time between downloading and viewing files and downloading and playing audio and video clips. We used both Netscape Communicator† and Internet Explorer† 4.0 (IE4). The first half consisted of repetitions of the following sequence:

- Browse with Netscape Communicator to www.irs.ustreas.gov/prod/cover.html, view several tax forms with Adobe Acrobat† Reader and download instructions for each form.
- Browse with Netscape Communicator to www.cse.ogi.edu and view a random (typically short) postscript TechReport with Ghostview†.
- Browse with IE4 to www.intel.com and view a processor manual with Adobe Acrobat Reader.
- Browse to www.research.microsoft.com with IE4 and view a technical report with Word 97.

In the second half we first browsed with Netscape Communicator to www.real.com and played news and music clips using RealPlayer†. We then browsed with IE 4.0 to Siskel and Ebert's Web site and played movie reviews using Shockwave†.

3.2 Test System Configuration

To minimize the impact of legacy software and hardware, such as Windows 98 drivers for devices on the old slow ISA (Industry Standard Architecture) bus, we configured our system exclusively with PCI (Peripherial Component Interconnect) bus and USB (Universal Serial Bus) devices. To do this we disabled the ISA Plug and Play Enumerator and motherboard ISA audio devices in the Control Panel System Properties menu (98) and Devices menu (NT).

Table 2 gives the full system configuration, with items that differ between the two systems shaded. The file systems used were different but reflect the "typical" file system for each OS. The audio solutions were of necessity different because Windows NT 4.0 does not support USB, while Windows 98 did not at the time fully support WDM audio drivers on PCI sound cards. A key point, easily overlooked, is that both OSs have been configured to use DMA drivers for the IDE devices (hard drive and CD-ROM). For Windows 98 this is a user configurable option accessible via the System icon on the Control Panel. For Windows NT 4.0 we used the Intel PIIXBus Master IDE Driver.

OS version	Windows NT 4.0	Windows 98
Optional OS Components	Service Pack 3 w. 11/97 rollup hotfix	Plus! 98† Pack w/o opt. Virus Scanner
Filesystem	NTFS	FAT32
IDE Driver	Intel PIIXBus Master IDE Drvr ver. 2.01.3	Default with DMA set ON
Processor & speed	Pentium® II 300 MHz	Pentium II 300 MHz
Motherboard	Atlanta (Intel 440 LX)	Atlanta (Intel 440 LX)
BIOS ver.	4A4LL0X0.86A.0 012.P02	4A4LL0X0.86A.0 012.P02
Memory	32 MB SDRAM	32 MB SDRAM
Hard Drive	Maxtor DiamondMax† 6.4 GB UDMA	Maxtor DiamondMax 6.4 GB UDMA
CD-ROM Drive	Sony CDU 711E 32x	Sony CDU 711E 32x
AGP Graphics	ATI Xpert@Work	ATI Xpert@Work
Resolution	1024 x 768 x 32bit	1024 x 768 x 32 bit
3D games	800 x 600 x 32 bit	800 x 600 x 32 bit
Audio solution	Ensoniq PCI sound card with Prosonic speakers	Phillips DSS 350 USB speakers
Network (only Web Browsing)	Intel EtherExpress™ Pro 100 PCI NIC	Intel EtherExpress Pro 100 PCI NIC

Table 2: Test System Configuration

4. Results

4.1 WDM scheduling hierarchy

For convenience, we abstract the services provided by the Win32 Driver Model (WDM) into an OS scheduling hierarchy as shown below. Each level of the OS scheduling hierarchy is fully preemptible by the level(s)

with lower numbers on Windows NT, but there are complications on Windows 98 since the legacy Windows 95 schedulers continue to exist[5]. In pure WDM (e.g., on Windows NT) the hierarchy is as follows:

1. Interrupt Service Routines (ISRs)
   Execute at IRQLs from `DIRQL` to `HighLevel`
2. WDM Deferred Procedure Calls (DPCs)
   FIFO/LIFO queue, three priorities (`High`, `Medium`, `Low Importance`), but DPCs cannot preempt other DPCs
3. Real-Time Priority Threads
   Timesliced, execute at Win32 priorities 16 through 31, can raise IRQL from PASSIVE (lowest) to arbitrarily high levels (i.e., block interrupts)
4. Normal Priority Threads
   Timesliced, execute at Win32 priorities 1 through 15, can raise IRQL

Our investigations focused on levels one through three of the scheduling hierarchy. We present data for the following:

- ISR latency for the PIT (timer) ISR.
- DPC latency for a "Medium Importance" WDM DPC enqueued by the PIT ISR. We term this the PIT DPC.
- High real-time priority kernel mode thread latency for a Win32 priority 28 kernel mode thread signaled by from the PIT DPC.
- Medium real-time priority kernel mode thread latency for a Win32 priority 24 kernel mode thread signaled from the PIT DPC.

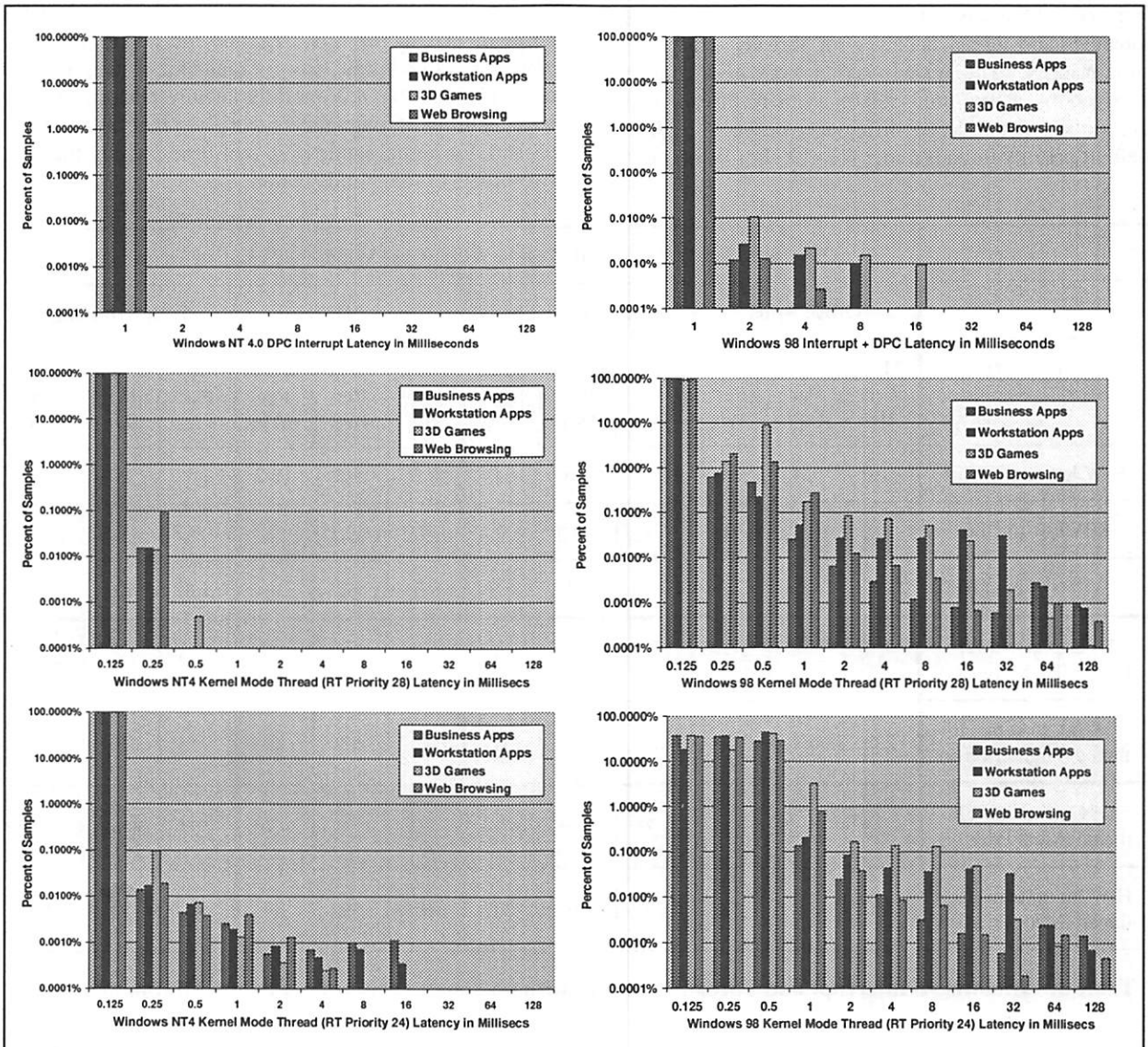


Figure 4: Measured Interrupt and Thread Latencies under Load on Windows NT 4.0 and Windows 98

4.2 Overall WDM Latency Profile

Windows 98 OS latency distributions are highly non-symmetric, with a very long tail on one side, and thus bear little resemblance to a normal distribution. In order to accurately show the latency tail we present our data as log-log plots in Figure 4. We present histograms comparing the latency from a timer interrupt (Programmable Interval Timer, whose ISR runs at extremely high IRQL) to the corresponding WDM DPC and from WDM DPCs to real-time high (28) and default (24) priority threads waiting on WDM synchronization events. We present this data broken out by application workload so that the latency profiles under different workloads on the same OS can be compared.

For NT 4.0 there is almost no distinction between DPC latencies and thread latencies for threads at high real-time priority. The WDM "kernel work item" queue is serviced by a real-time default priority thread, which accounts for the large difference between high and default priority threads under NT 4.0. For Windows 98, on the other hand, there is an order of magnitude reduction in worst-case latencies that a driver obtains by using WDM DPCs as opposed to real-time priority kernel mode threads. NT real-time high priority threads and DPCs exhibit worst-case latencies which are an order of magnitude lower than those of Windows 98 DPCs and Windows NT real-time default priority threads. This view of system performance contrasts sharply with the view one obtains using traditional throughput-based benchmarks.

To verify that throughput-based benchmarks would not reveal the variation in real-time performance that we see in our plots, we ran the Business Winstone 97 benchmark on Windows 98 and on Windows NT 4.0 using our system configurations as specified in Table 2. While reporting requirements (and space here) prevent us from publishing exact figures, the average delta between like scores was 10% and the maximum delta was 20%. In contrast, from a real-time standpoint, we conclude that NT 4.0 exhibits latency performance at least an order of magnitude superior to that of Windows 98 and, for kernel mode high real-time priority threads, two orders of magnitude better.

	Observed Hourly, Daily and Weekly Worst Case Windows 98 Latencies (in ms.)											
	Office Apps			**Workstation Apps**			**Recent 3D Games**			**Web Browsing**		
OS Service	**Max Per Hr**	**Max Per Day**	**Max Per Wk**	**Max Per Hr**	**Max Per Day**	**Max Per Wk**	**Max Per Hr**	**Max Per Day**	**Max Per Wk**	**Max Per Hr**	**Max Per Day**	**Max Per Wk**
H/W Int. to S/W ISR	<1.0	1.4	1.6	2.2	5.6	6.3	8.8	9.7	12.2	1.1	1.7	3.5
S/W ISR to DPC	+ 0.1	+ 0.1	+ 0.4	+ 0.5	+ 0.5	+ 0.6	+ 0.9	+ 2.1	+ 2.1	+ 0.2	+ 0.3	+ 0.3
H/W Interrupt to DPC	1.0	1.5	2.0	2.7	6.1	6.9	9.7	12	14	1.3	2.0	3.8
DPC to kernel RT thread (High Priority)	+ 1.6	+ 5.2	+ 31	+ 21	+ 24	+ 24	+ 35	+ 46	+ 70	+ 14	+ 68	+ 80
H/W Int. to kernel RT thread (High Priority)	2.6	6.7	33	24	30	31	45	58	84	15	70	84
DPC to kernel RT thread (Med. Priority)	+ 3.1	+ 6.7	+ 31	+ 21	+ 23	+ 24	+ 36	+ 47	+ 70	+ 51	+ 68	+ 80
H/W Int. to kernel RT thread (Med. Priority)	4.1	8.2	33	24	29	31	46	59	84	52	70	84

Table 3: Windows 98 Interrupt and Thread Latencies with no Sound Scheme on a PC 99 Minimum System

4.3 Windows 98 Detailed Latency Profile

Because Windows 98 has been recently released, we present a more detailed latency profile in tabular form. Since our focus is on means of forecasting realizable application worst-case behavior, we are especially concerned with the worst-case latency and with comparative measures of the "thickness" of the tail of the latency distribution. We therefore characterized the distributions for Windows 98 in terms of three expected worst case values: hourly, daily and weekly. The hourly value is for continuous usage, whereas the daily and weekly values do *not* represent continuous 24 or 168 hour usage, but rather expected average daily and weekly use by a heavy user. The usage patterns are described in detail in section 3.1. To briefly recap, for the Office and Workstation applications a "day" is 6 to 8 hours long and a (work) week has 5 "days", while for the 3D games and Web Browsing, a "day" is only 3 to 4 hours but a (consumer) week has 7 "days".

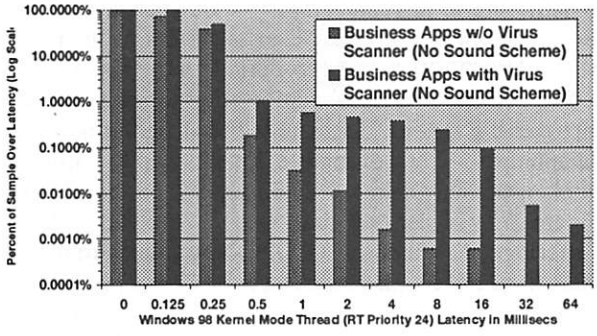


Figure 5: Effect of the Virus Scanner on High Priority Real-Time Thread Latency

During the course of our investigation of Windows 98 we discovered the optional Plus! 98[†] Pack Virus Scanner and the Windows sound schemes had significant impacts on thread latency. The Virus Scanner is particularly egregious in this regard and the data for Windows 98 presented in Figure 4 is for an installation without the virus scanner. Figure 5 presents data with the virus scanner installed and active, but with no sound scheme, and it can be seen that with the virus scanner 16 millesecond thread latencies occur over two orders of magnitude more frequently. Assuming that long latencies are uniformly distributed over time, with the virus scanner on we would expect a 16 millisecond thread latency about every 1000 times that our thread does a `WaitForSingleObject` on a WDM event, or roughly every 16 seconds for an audio thread with a 16 millisecond period. In contrast, without the virus scanner (and with no sound scheme) we would expect a 16 millisecond thread latency only about once in 165,000 waits, or roughly once every 44 minutes for the same audio thread. Intel's audio experts did not find it surprising that the virus scanner had this effect; they had remarked for some time that the virus scanner causes breakup of low latency audio.

4.4 Windows 98 Thread Latency Causes

As we have seen in the previous section, our tools can successfully predict Quality of Service problems that impact the end user's experience. More importantly, by defining a simple metric that is easy to automatically detect at run-time, our tools and techniques give us the ability to determine the code paths that are responsible for this behavior. This, of course, greatly increases the probability of obtaining a fix from the developers, who now receive a bug of the form "audio breaks up when we turn on your application", but who could receive a bug report with one or more function call traces.

Before discussing more specific results, some background is in order. The Windows 98 Plus! Pack makes a number of sound schemes available. These produce a variety of user-selectable sounds upon occurrence of various "events". These "events" range from typical things, such as popup of a Dialog Box to the more esoteric, such as traversal of walking menus (i.e., EVERY time a submenu appears). As mentioned above, Winstone uses MS-Test to drive applications at greater than human speeds, which results in a lot of sounds being played. During our testing we restricted ourselves to the default and "no sound" sound schemes.

```
Analysis of latency episode number 0
   1 samples in VMM function @KfLowerIrqI
   1 samples in NTKERN function
     _ExpAllocatePool
   1 samples in SYSAUDIO function
     _ProcessTopologyConnection
   2 samples in VMM function
     _mmCalcFrameBadness
   ------------------------------------------------
   5 total samples in episode

Analysis of latency episode number 1
   1 samples in SYSAUDIO function
     _ProcessTopologyConnection
   2 samples in VMM function
     _mmCalcFrameBadness
   2 samples in VMM function _mmFindContig
   1 samples in KMIXER function unknown
   ------------------------------------------------
   6 total samples in episode
```

Table 4: Thread Latency Cause Tool Output, Windows 98 w. Biz Apps, Default Sound Scheme

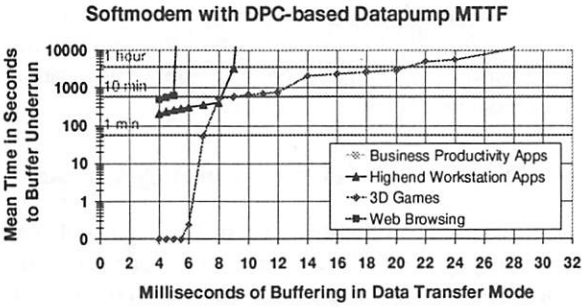


Figure 6: Mean Time to Buffer Underrun for a DPC-based Datapump of a Soft Modem on Windows 98 in Data Transfer Mode

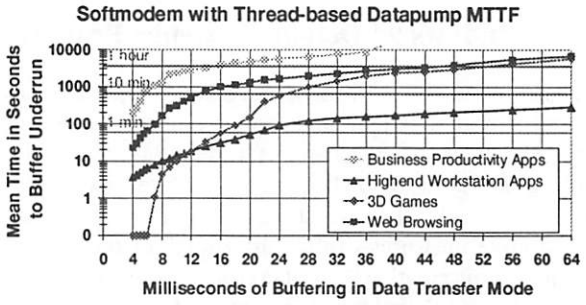


Figure 7: Mean Time to Buffer Underrun for a Thread-based Datapump of a Softmodem on Windows 98 in Data Transfer Mode

Table 4 presents two brief sample traces from an investigation into the causes of long thread latencies during the Winstone Business benchmark when the default Windows sound scheme was enabled. From the traces we see that with the default sound scheme on (presumably the normal state of affairs) two moderately long thread thread latencies were observed. During both a SysAudio function ProcessTopologyConnection was active and the OS appears to have been allocating contiguous memory, possibly in order to accommodate "bad", possibly misaligned, audio frames. We can also see that at least part of this operation is taking place at raised IRQL, which would explain, for example, why both priority 24 and 28 kernel mode threads are affected. Further analysis of these episodes is best left to the authors of the code, but the reader will see that this information can be of great use.

5. Analysis

As an example of how detailed latency data can be used to forecast quality of service for multimedia applications and low latency drivers, we present a brief analysis of soft modem quality of service as a function of the size and the number of buffers (and thus, the allowable latency in servicing the buffers). Here we briefly discuss the Mean Time To Failure (MTTF) plots that we present in the next section. The plots are derived from our tables of latency data by calculating the slack time for each amount of buffering (i.e., $t * (n-1) - c$, where n is the number of buffers, t is the buffer size in milliseconds and c is the compute time for 1 buffer.). This number is used to index into the latency table to determine the frequency with which such latencies occur, and this frequency is divided by an approximation of the cycle time (for simplicity, $(n-1) * t$). Thus the calculation is strictly accurate only for double buffered implementations but is reasonably accurate if n is small.

5.1 Soft Modem Quality of Service

We now present an analysis of soft modem quality of service from a timing standpoint. Figures 6 and 7 show the mean time to failure (i.e., buffer underrun) for the datapump of a soft modem as a function of the amount of buffering in the datapump. We have estimated that the datapump requires 25% of a system with a 300 MHz Pentium II processor during data transmission mode, which is a conservative (high) estimate. To interpret the figures, calculate the total buffering in the datapump. For example, for a triple buffered implementation using 6 millisecond buffers, using Figure 6, we can see that with 12 milliseconds of buffering the Windows 98 DPC-based datapump will miss a buffer roughly once every 12 to 15 minutes (720-900 seconds) while playing an "average" 3D game. With 10 millisecond buffers triple buffered (i.e., 20 milliseconds of buffering), however, the Windows 98 DPC-based datapump would average an hour (3600 seconds) between misses while playing an "average" 3D game[6]. Similarly, a Windows 98 thread-based datapump that uses high-priority, real-time kernel mode threads will require about 48 milliseconds of latency tolerance (e.g., four 16 millisecond buffers) in order to average an hour between misses while playing an

[6] Note that missed buffers need not be catastrophic since design techniques exist whereby the datapump can arrange for hardware to automatically transmit a dummy buffer in the event that a buffer is missed. Such a dummy buffer can be made indistinguishable from line noise or other bit errors to the receiving modem and will result in either a retransmission request at the link layer or a dropped frame. In either case the overall impact on modem connection quality can be kept manageably small relative to the number of buffers which will be damaged due to line noise. Similar considerations (e.g., error correcting codes) apply to other classes of low latency real-time drivers.

"average" 3D game. Since the worst case latencies for Windows NT are uniformly below the minimum modem slack time of 3 milliseconds (= cycle time of 4 ms. – 1 ms. of computation), we forgo the analysis.

5.2 Schedulability Analysis on a Non-Real-Time OS

As we noted above, the MTTF plots in the previous section assume implicitly that double buffering is used, but are reasonably accurate for triple buffering. A more accurate method of making the assessment, including taking into account other "lower level" (i.e., higher priority) drivers that were not present on the measured, is described in our earlier work on Schedulability Analysis [4]. Briefly, the procedure is to use the information from Table 3 as input to a Schedulability Analysis tool. One chooses the worst case latency as a function of the permissible error rate: for example, one dropped buffer every five or ten minutes for low latency audio (video teleconferencing), one dropped buffer per hour for a soft modem, or one dropped buffer per day for a more high-reliability device. The worst-case is then used to calculate a "pseudo worst-case" which is input into a standard schedulability analysis tool such as PERTS [16]. This technique amortizes the overhead of an unusually long latency over a number of "average" latencies to enable analysis techniques designed for deterministic real-time OSs to be applied on a general purpose OS.

6. Conclusions

We have presented a metric for evaluating the real-time performance of non-real-time OSs and platforms. This metric captures an aspect of performance that is completely missed by standard batch and throughput-based benchmarking techniques commonly in use today. The techniques that we have described are destined to grow in importance as emerging workloads such as audio, video and other multimedia presentations are ever more widely deployed and as low latency hard real-time drivers are migrated off of special purpose hardware onto host processors. This process is already well advanced, with applications such as soft MPEG and DVD already under development and soft audio and soft modems already being routinely deployed by vendors of low-cost personal computers. It is likely that this trend will accelerate in the future, further increasing the importance of the latency metric.

Our analysis revealed that the two implementations of the Windows Driver Model, although functionally compatible, are very different in their timing behavior. Using the interrupt and thread latency metrics we are able to characterize the behavior that applications and drivers will experience on Windows 98 even before those applications and drivers are fielded. Our analysis indicates that many compute-intensive drivers will be forced to use DPCs on Windows 98, whereas on Windows NT high-priority, real-time kernel mode threads should provide service indistinguishable from DPCs for all but the most demanding low latency drivers. When one considers the difficulties of "interrupt-level" (i.e., WDM DPCs) driver development and the multitude of benefits obtained from using threads, it is apparent that analyses such as the one we have just presented will become increasingly important from a Software Engineering standpoint.

6.1 Future Work

We have completed evaluations of Windows 98 [5] and Windows NT 4.0 and continue to monitor the performance of Beta releases of Windows 2000[7]. We have also developed a tool that models periodic computation at configurable modalities (e.g., threads, DPCs) and priorities within modalities, and reports the number of deadlines that have been missed. With this tool we can model a soft modem and examine its impact on other kernel mode services. We will also be able to use the tool to validate our quality of service predictions in this paper and expect to report on this work at the conference.

In addition to this work, the latency cause analysis tool is under active development. First, we plan to enhance it to hook non-maskable interrupts caused by the Pentium II performance monitoring counters instead of the PIT interrupt. By configuring the performance counter to the CPU_CLOCKS_UNHALTED event we will be able to get sub-millisecond resolution during both thread and interrupt latencies. Second, we would like to enhance the hook to "walk" the stack so as to generate call trees instead of isolated instruction pointer samples. This would give much more visibility into the actual code paths under execution, greatly increasing the utility of the data.

7. Acknowledgements

Venugopal Padmanabhan and Dorian Salcz collected the lab data; their patience and precision are gratefully acknowledged. Dan Cox provided managerial support and encouragement. Others who assisted at various times include Dan Baumberger, Lyle Cool, Barbara Denniston, Tom Dingwall, Judi Goldstein, Jaya Jeyaseelan, Dan Nowlin, Barry O'Mahony, Jeyashree Padmanabhan, Jeff Spruiel, Jim Stanley, Cindy Ward,

[7] Windows 2000 was previously Windows NT 5.0.

and Mary Griffith. In addition, the support of Tom Barnes, Darin Eames and Sanjay Panditji of the Intel Architecture Labs is gratefully noted, as is the patience of the Program Committee shepherd, Margo Seltzer. Finally, Erik's wife, Judy, was exceptionally patient during the period when this paper was being written.

8. References

[1] A. Baker, *The Windows NT Device Driver Book*, Prentice Hall, Upper Saddle River, NJ. 1997.

[2] B. N. Bershad, R.P. Draves and A. Forin, "Using Microbenchmarks to Evaluate System Performance", *Proc. 3rd Wkshop on Workstation Operating Systems*, Key Biscayne, FL, April, 1992

[3] A.B. Brown and M.I. Seltzer, "Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture", *Proc. 1997 Sigmetrics Conf.*, Seattle, WA, June 1997.

[4] E. Cota-Robles, J. Held and T. J. Barnes, "Schedulability Analysis for Desktop Multimedia Applications: Simple Ways to Handle General-Purpose Operating Systems and Open Environments", *Proc. 4th IEEE International Conf. on Multimedia Computing and Systems*, Ottawa, Canada, June 1997. URL: http://developer.intel.com/ial/sm/doc.htm

[5] E. Cota-Robles, "Implications of Windows OS Latency for WDM Drivers", *Intel Developer's Forum*, Palm Springs, CA, September 1998.

[6] E. Cota-Robles, "Windows 98 Latency Characterization for WDM Kernel Drivers", *Intel Architecture Lab White Paper*, July 1998. URL: http://developer.intel.com/ial/sm/doc.htm

[7] Y. Endo, Z. Wang, J. B. Chen and M. I. Seltzer, " Using Latency to Evaluate Interactive System Performance", *Proc. of the Second Symp. on Operating Systems Design and Implementation*, Seattle, WA, October 1996.

[8] Y. Endo, M. I. Seltzer, "Measuring Windows NT—Possibilities and Limitations", *First USENIX Windows NT Workshop*, Seattle, WA, Aug, 1997.

[9] Intel Corp., *Intel Architecture Software Developer's Manual*, 3 volumes, 1996. URL: http://developer.intel.com/design/intarch/manuals/index.htm

[10] Intel Corp., *Pentium® II Processor Developer's Manual*, 1997. URL: http://developer.intel.com/design/PentiumII/manuals/index.htm

[11] International Telecommunication Union. Draft Reccomendation G.992.2, Splitterless Asymetrical Digital Subscriber Line (ADSL) Transceivers, 1998

[12] M Jones, D. Regehr. "Issues in using commodity operating systems for time-dependent tasks: experiences from a study of Windows NT", Proc. of 8th Intl Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 98), Cambridge, U.K., July 1998.

[13] D. I. Katcher, H. Arakawa, and J. Strosnider. "Engineering and Analysis of Fixed Priority Schedulers", *IEEE Transactions on Software Engineering*, 19(9), September, 1993.

[14] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practioner's Handbook for Real-Time Analysis*. Kluwer, Boston, MA, 1993.

[15] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment", *JACM*, 20(1), Jan, 1973.

[16] J.W.S. Liu, J.L. Redondo, Z. Deng, T.S. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha and W.K. Shih, " PERTS: A Prototyping Environment for Real-Time Systems". *Proc. of the IEEE Real-Time Systems Symposium*, December 1993.

[17] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis", *Proc. 1996 USENIX Technical Conf.*, San Diego, CA, January, 1996.

[18] Microsoft Corporation, "Windows 98 Driver Development Kit (DDK)" in Microsoft Developer Network Prof. Edition, Redmond, WA, 1998.

[19] J. K. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast as Hardware", *Proc. of the USENIX Summer Conf.*, June, 1990.

[20] B. Shneiderman, *Designing the User Interface: Strategies of Effective Human-Computer Interaction*. Addison-Wesley, Reading, MA. 1992

[21] D. A. Solomon, *Inside Windows NT Second Edition*, Microsoft Press, Redmond, WA. 1998.

[22] Ziff-Davis Corp., "Labs Notes: Benchmark 97: Inside PC Labs' Latest Tests", *PC Magazine Online*, Vol. 15, No. 21, December 3, 1996.

[23] Ziff-Davis Corp., Winstone webmaster, personal communication, 1998.

Practical Byzantine Fault Tolerance

Miguel Castro and Barbara Liskov
Laboratory for Computer Science,
Massachusetts Institute of Technology,
545 Technology Square, Cambridge, MA 02139
{castro,liskov}@lcs.mit.edu

Abstract

This paper describes a new replication algorithm that is able to tolerate Byzantine faults. We believe that Byzantine-fault-tolerant algorithms will be increasingly important in the future because malicious attacks and software errors are increasingly common and can cause faulty nodes to exhibit arbitrary behavior. Whereas previous algorithms assumed a synchronous system or were too slow to be used in practice, the algorithm described in this paper is practical: it works in asynchronous environments like the Internet and incorporates several important optimizations that improve the response time of previous algorithms by more than an order of magnitude. We implemented a Byzantine-fault-tolerant NFS service using our algorithm and measured its performance. The results show that our service is only 3% slower than a standard unreplicated NFS.

1 Introduction

Malicious attacks and software errors are increasingly common. The growing reliance of industry and government on online information services makes malicious attacks more attractive and makes the consequences of successful attacks more serious. In addition, the number of software errors is increasing due to the growth in size and complexity of software. Since malicious attacks and software errors can cause faulty nodes to exhibit Byzantine (i.e., arbitrary) behavior, Byzantine-fault-tolerant algorithms are increasingly important.

This paper presents a new, *practical* algorithm for state machine replication [17, 34] that tolerates Byzantine faults. The algorithm offers both liveness and safety provided at most $\lfloor \frac{n-1}{3} \rfloor$ out of a total of n replicas are simultaneously faulty. This means that clients eventually receive replies to their requests and those replies are correct according to linearizability [14, 4]. The algorithm works in asynchronous systems like the Internet and it incorporates important optimizations that enable it to perform efficiently.

There is a significant body of work on agreement and replication techniques that tolerate Byzantine faults (starting with [19]). However, most earlier work (e.g., [3, 24, 10]) either concerns techniques designed to demonstrate theoretical feasibility that are too inefficient to be used in practice, or assumes synchrony, i.e., relies on known bounds on message delays and process speeds. The systems closest to ours, Rampart [30] and SecureRing [16], were designed to be practical, but they rely on the synchrony assumption for correctness, which is dangerous in the presence of malicious attacks. An attacker may compromise the safety of a service by delaying non-faulty nodes or the communication between them until they are tagged as faulty and excluded from the replica group. Such a denial-of-service attack is generally easier than gaining control over a non-faulty node.

Our algorithm is not vulnerable to this type of attack because it does not rely on synchrony for safety. In addition, it improves the performance of Rampart and SecureRing by more than an order of magnitude as explained in Section 7. It uses only one message round trip to execute read-only operations and two to execute read-write operations. Also, it uses an efficient authentication scheme based on message authentication codes during normal operation; public-key cryptography, which was cited as the major latency [29] and throughput [22] bottleneck in Rampart, is used only when there are faults.

To evaluate our approach, we implemented a replication library and used it to implement a real service: a Byzantine-fault-tolerant distributed file system that supports the NFS protocol. We used the Andrew benchmark [15] to evaluate the performance of our system. The results show that our system is only 3% slower than the standard NFS daemon in the Digital Unix kernel during normal-case operation.

Thus, the paper makes the following contributions:

- It describes the first state-machine replication protocol that correctly survives Byzantine faults in asynchronous networks.
- It describes a number of important optimizations that allow the algorithm to perform well so that it can be used in real systems.

This research was supported in part by DARPA under contract DABT63-95-C-005, monitored by Army Fort Huachuca, and under contract F30602-98-1-0237, monitored by the Air Force Research Laboratory, and in part by NEC. Miguel Castro was partially supported by a PRAXIS XXI fellowship.

- It describes the implementation of a Byzantine-fault-tolerant distributed file system.
- It provides experimental results that quantify the cost of the replication technique.

The remainder of the paper is organized as follows. We begin by describing our system model, including our failure assumptions. Section 3 describes the problem solved by the algorithm and states correctness conditions. The algorithm is described in Section 4 and some important optimizations are described in Section 5. Section 6 describes our replication library and how we used it to implement a Byzantine-fault-tolerant NFS. Section 7 presents the results of our experiments. Section 8 discusses related work. We conclude with a summary of what we have accomplished and a discussion of future research directions.

2 System Model

We assume an asynchronous distributed system where nodes are connected by a network. The network may fail to deliver messages, delay them, duplicate them, or deliver them out of order.

We use a Byzantine failure model, i.e., faulty nodes may behave arbitrarily, subject only to the restriction mentioned below. We assume independent node failures. For this assumption to be true in the presence of malicious attacks, some steps need to be taken, e.g., each node should run different implementations of the service code and operating system and should have a different root password and a different administrator. It is possible to obtain different implementations from the same code base [28] and for low degrees of replication one can buy operating systems from different vendors. N-version programming, i.e., different teams of programmers produce different implementations, is another option for some services.

We use cryptographic techniques to prevent spoofing and replays and to detect corrupted messages. Our messages contain public-key signatures [33], message authentication codes [36], and message digests produced by collision-resistant hash functions [32]. We denote a message m signed by node i as $\langle m \rangle_{\sigma_i}$ and the digest of message m by $D(m)$. We follow the common practice of signing a digest of a message and appending it to the plaintext of the message rather than signing the full message ($\langle m \rangle_{\sigma_i}$ should be interpreted in this way). All replicas know the others' public keys to verify signatures.

We allow for a very strong adversary that can coordinate faulty nodes, delay communication, or delay correct nodes in order to cause the most damage to the replicated service. We do assume that the adversary cannot delay correct nodes indefinitely. We also assume that the adversary (and the faulty nodes it controls) are computationally bound so that (with very high probability) it is unable to subvert the cryptographic techniques mentioned above. For example, the adversary cannot produce a valid signature of a non-faulty node, compute the information summarized by a digest from the digest, or find two messages with the same digest. The cryptographic techniques we use are thought to have these properties [33, 36, 32].

3 Service Properties

Our algorithm can be used to implement any deterministic replicated *service* with a *state* and some *operations*. The operations are not restricted to simple reads or writes of portions of the service state; they can perform arbitrary deterministic computations using the state and operation arguments. Clients issue requests to the replicated service to invoke operations and block waiting for a reply. The replicated service is implemented by n replicas. Clients and replicas are non-faulty if they follow the algorithm in Section 4 and if no attacker can forge their signature.

The algorithm provides both *safety* and *liveness* assuming no more than $\lfloor \frac{n-1}{3} \rfloor$ replicas are faulty. Safety means that the replicated service satisfies linearizability [14] (modified to account for Byzantine-faulty clients [4]): it behaves like a centralized implementation that executes operations atomically one at a time. Safety requires the bound on the number of faulty replicas because a faulty replica can behave arbitrarily, e.g., it can destroy its state.

Safety is provided regardless of how many faulty clients are using the service (even if they collude with faulty replicas): all operations performed by faulty clients are observed in a consistent way by non-faulty clients. In particular, if the service operations are designed to preserve some invariants on the service state, faulty clients cannot break those invariants.

The safety property is insufficient to guard against faulty clients, e.g., in a file system a faulty client can write garbage data to some shared file. However, we limit the amount of damage a faulty client can do by providing access control: we authenticate clients and deny access if the client issuing a request does not have the right to invoke the operation. Also, services may provide operations to change the access permissions for a client. Since the algorithm ensures that the effects of access revocation operations are observed consistently by all clients, this provides a powerful mechanism to recover from attacks by faulty clients.

The algorithm does not rely on synchrony to provide safety. Therefore, it must rely on synchrony to provide liveness; otherwise it could be used to implement consensus in an asynchronous system, which is not possible [9]. We guarantee liveness, i.e., clients eventually receive replies to their requests, provided at most $\lfloor \frac{n-1}{3} \rfloor$ replicas are faulty and $delay(t)$ does not

grow faster than 2^t indefinitely. Here, $delay(t)$ is the time between the moment t when a message is sent for the first time and the moment when it is received by its destination (assuming the sender keeps retransmitting the message until it is received). (A more precise definition can be found in [4].) This is a rather weak synchrony assumption that is likely to be true in any real system provided network faults are eventually repaired, yet it enables us to circumvent the impossibility result in [9].

The resiliency of our algorithm is optimal: $3f+1$ is the minimum number of replicas that allow an asynchronous system to provide the safety and liveness properties when up to f replicas are faulty (see [2] for a proof). This many replicas are needed because it must be possible to proceed after communicating with $n - f$ replicas, since f replicas might be faulty and not responding. However, it is possible that the f replicas that did not respond are not faulty and, therefore, f of those that responded might be faulty. Even so, there must still be enough responses that those from non-faulty replicas outnumber those from faulty ones, i.e., $n - 2f > f$. Therefore $n > 3f$.

The algorithm does not address the problem of fault-tolerant privacy: a faulty replica may leak information to an attacker. It is not feasible to offer fault-tolerant privacy in the general case because service operations may perform arbitrary computations using their arguments and the service state; replicas need this information in the clear to execute such operations efficiently. It is possible to use secret sharing schemes [35] to obtain privacy even in the presence of a threshold of malicious replicas [13] for the arguments and portions of the state that are opaque to the service operations. We plan to investigate these techniques in the future.

4 The Algorithm

Our algorithm is a form of *state machine* replication [17, 34]: the service is modeled as a state machine that is replicated across different nodes in a distributed system. Each state machine replica maintains the service state and implements the service operations. We denote the set of replicas by $\mathcal{R}$ and identify each replica using an integer in $\{0, ..., |\mathcal{R}| - 1\}$. For simplicity, we assume $|\mathcal{R}| = 3f + 1$ where f is the maximum number of replicas that may be faulty; although there could be more than $3f + 1$ replicas, the additional replicas degrade performance (since more and bigger messages are being exchanged) without providing improved resiliency.

The replicas move through a succession of configurations called *views*. In a view one replica is the *primary* and the others are *backups*. Views are numbered consecutively. The primary of a view is replica p such that $p = v \ mod \ |\mathcal{R}|$, where v is the view number. View changes are carried out when it appears that the primary has failed. Viewstamped Replication [26] and Paxos [18] used a similar approach to tolerate benign faults (as discussed in Section 8.)

The algorithm works roughly as follows:

1. A client sends a request to invoke a service operation to the primary
2. The primary multicasts the request to the backups
3. Replicas execute the request and send a reply to the client
4. The client waits for $f + 1$ replies from different replicas with the same result; this is the result of the operation.

Like all state machine replication techniques [34], we impose two requirements on replicas: they must be *deterministic* (i.e., the execution of an operation in a given state and with a given set of arguments must always produce the same result) and they must start in the same state. Given these two requirements, the algorithm ensures the safety property by guaranteeing that *all non-faulty replicas agree on a total order for the execution of requests despite failures.*

The remainder of this section describes a simplified version of the algorithm. We omit discussion of how nodes recover from faults due to lack of space. We also omit details related to message retransmissions. Furthermore, we assume that message authentication is achieved using digital signatures rather than the more efficient scheme based on message authentication codes; Section 5 discusses this issue further. A detailed formalization of the algorithm using the I/O automaton model [21] is presented in [4].

4.1 The Client

A client c requests the execution of state machine operation o by sending a $\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$ message to the primary. Timestamp t is used to ensure *exactly-once* semantics for the execution of client requests. Timestamps for c's requests are totally ordered such that later requests have higher timestamps than earlier ones; for example, the timestamp could be the value of the client's local clock when the request is issued.

Each message sent by the replicas to the client includes the current view number, allowing the client to track the view and hence the current primary. A client sends a request to what it believes is the current primary using a point-to-point message. The primary atomically multicasts the request to all the backups using the protocol described in the next section.

A replica sends the reply to the request directly to the client. The reply has the form $\langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma_i}$ where v is the current view number, t is the timestamp of the corresponding request, i is the replica number, and r is the result of executing the requested operation.

The client waits for $f + 1$ replies with valid signatures from different replicas, and with the same t and r, before

accepting the result r. This ensures that the result is valid, since at most f replicas can be faulty.

If the client does not receive replies soon enough, it broadcasts the request to all replicas. If the request has already been processed, the replicas simply re-send the reply; replicas remember the last reply message they sent to each client. Otherwise, if the replica is not the primary, it relays the request to the primary. If the primary does not multicast the request to the group, it will eventually be suspected to be faulty by enough replicas to cause a view change.

In this paper we assume that the client waits for one request to complete before sending the next one. But we can allow a client to make asynchronous requests, yet preserve ordering constraints on them.

4.2 Normal-Case Operation

The state of each replica includes the state of the service, a *message log* containing messages the replica has accepted, and an integer denoting the replica's current view. We describe how to truncate the log in Section 4.3.

When the primary, p, receives a client request, m, it starts a three-phase protocol to atomically multicast the request to the replicas. The primary starts the protocol immediately unless the number of messages for which the protocol is in progress exceeds a given maximum. In this case, it buffers the request. Buffered requests are multicast later as a group to cut down on message traffic and CPU overheads under heavy load; this optimization is similar to a group commit in transactional systems [11]. For simplicity, we ignore this optimization in the description below.

The three phases are *pre-prepare, prepare,* and *commit*. The pre-prepare and prepare phases are used to totally order requests sent in the same view even when the primary, which proposes the ordering of requests, is faulty. The prepare and commit phases are used to ensure that requests that commit are totally ordered across views.

In the pre-prepare phase, the primary assigns a sequence number, n, to the request, multicasts a pre-prepare message with m piggybacked to all the backups, and appends the message to its log. The message has the form $\langle\langle \text{PRE-PREPARE}, v, n, d\rangle_{\sigma_p}, m\rangle$, where v indicates the view in which the message is being sent, m is the client's request message, and d is m's digest.

Requests are not included in pre-prepare messages to keep them small. This is important because pre-prepare messages are used as a proof that the request was assigned sequence number n in view v in view changes. Additionally, it decouples the protocol to totally order requests from the protocol to transmit the request to the replicas; allowing us to use a transport optimized for small messages for protocol messages and a transport optimized for large messages for large requests.

A backup accepts a pre-prepare message provided:

- the signatures in the request and the pre-prepare message are correct and d is the digest for m;
- it is in view v;
- it has not accepted a pre-prepare message for view v and sequence number n containing a different digest;
- the sequence number in the pre-prepare message is between a low water mark, h, and a high water mark, H.

The last condition prevents a faulty primary from exhausting the space of sequence numbers by selecting a very large one. We discuss how H and h advance in Section 4.3.

If backup i accepts the $\langle\langle \text{PRE-PREPARE}, v, n, d\rangle_{\sigma_p}, m\rangle$ message, it enters the *prepare* phase by multicasting a $\langle \text{PREPARE}, v, n, d, i\rangle_{\sigma_i}$ message to all other replicas and adds both messages to its log. Otherwise, it does nothing.

A replica (including the primary) accepts prepare messages and adds them to its log provided their signatures are correct, their view number equals the replica's current view, and their sequence number is between h and H.

We define the predicate *prepared*(m, v, n, i) to be true if and only if replica i has inserted in its log: the request m, a pre-prepare for m in view v with sequence number n, and $2f$ prepares from different backups that match the pre-prepare. The replicas verify whether the prepares match the pre-prepare by checking that they have the same view, sequence number, and digest.

The pre-prepare and prepare phases of the algorithm guarantee that non-faulty replicas agree on a total order for the requests within a view. More precisely, they ensure the following invariant: if *prepared*(m, v, n, i) is true then *prepared*(m', v, n, j) is false for any non-faulty replica j (including $i = j$) and any m' such that $D(m') \neq D(m)$. This is true because *prepared*(m, v, n, i) and $|\mathcal{R}| = 3f + 1$ imply that at least $f + 1$ non-faulty replicas have sent a pre-prepare or prepare for m in view v with sequence number n. Thus, for *prepared*(m', v, n, j) to be true at least one of these replicas needs to have sent two conflicting prepares (or pre-prepares if it is the primary for v), i.e., two prepares with the same view and sequence number and a different digest. But this is not possible because the replica is not faulty. Finally, our assumption about the strength of message digests ensures that the probability that $m \neq m'$ and $D(m) = D(m')$ is negligible.

Replica i multicasts a $\langle \text{COMMIT}, v, n, D(m), i\rangle_{\sigma_i}$ to the other replicas when *prepared*(m, v, n, i) becomes true. This starts the commit phase. Replicas accept commit messages and insert them in their log provided they are properly signed, the view number in the message is equal to the replica's current view, and the sequence number is between h and H

We define the *committed* and *committed-local* predicates as follows: *committed*(m, v, n) is true if and only if *prepared*(m, v, n, i) is true for all i in some set of $f+1$ non-faulty replicas; and *committed-local*(m, v, n, i) is true if and only if *prepared*(m, v, n, i) is true and i has accepted $2f + 1$ commits (possibly including its own) from different replicas that match the pre-prepare for m; a commit matches a pre-prepare if they have the same view, sequence number, and digest.

The commit phase ensures the following invariant: if *committed-local*(m, v, n, i) is true for some non-faulty i then *committed*(m, v, n) is true. This invariant and the view-change protocol described in Section 4.4 ensure that non-faulty replicas agree on the sequence numbers of requests that commit locally even if they commit in different views at each replica. Furthermore, it ensures that any request that commits locally at a non-faulty replica will commit at $f + 1$ or more non-faulty replicas eventually.

Each replica i executes the operation requested by m after *committed-local*(m, v, n, i) is true and i's state reflects the sequential execution of all requests with lower sequence numbers. This ensures that all non-faulty replicas execute requests in the same order as required to provide the safety property. After executing the requested operation, replicas send a reply to the client. Replicas discard requests whose timestamp is lower than the timestamp in the last reply they sent to the client to guarantee exactly-once semantics.

We do not rely on ordered message delivery, and therefore it is possible for a replica to commit requests out of order. This does not matter since it keeps the pre-prepare, prepare, and commit messages logged until the corresponding request can be executed.

Figure 1 shows the operation of the algorithm in the normal case of no primary faults. Replica 0 is the primary, replica 3 is faulty, and C is the client.

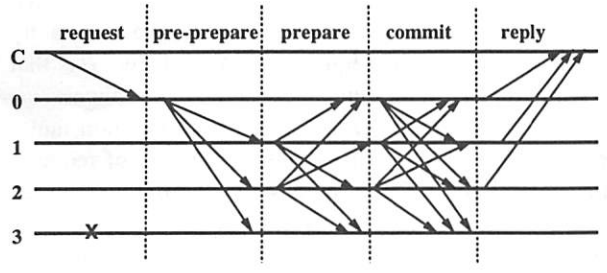


Figure 1: Normal Case Operation:

4.3 Garbage Collection

This section discusses the mechanism used to discard messages from the log. For the safety condition to hold, messages must be kept in a replica's log until it knows that the requests they concern have been executed by at least $f + 1$ non-faulty replicas and it can prove this to others in view changes. In addition, if some replica misses messages that were discarded by all non-faulty replicas, it will need to be brought up to date by transferring all or a portion of the service state. Therefore, replicas also need some proof that the state is correct.

Generating these proofs after executing every operation would be expensive. Instead, they are generated periodically, when a request with a sequence number divisible by some constant (e.g., 100) is executed. We will refer to the states produced by the execution of these requests as *checkpoints* and we will say that a checkpoint with a proof is a *stable checkpoint*.

A replica maintains several logical copies of the service state: the last stable checkpoint, zero or more checkpoints that are not stable, and a current state. Copy-on-write techniques can be used to reduce the space overhead to store the extra copies of the state, as discussed in Section 6.3.

The proof of correctness for a checkpoint is generated as follows. When a replica i produces a checkpoint, it multicasts a message $\langle \text{CHECKPOINT}, n, d, i \rangle_{\sigma_i}$ to the other replicas, where n is the sequence number of the last request whose execution is reflected in the state and d is the digest of the state. Each replica collects checkpoint messages in its log until it has $2f + 1$ of them for sequence number n with the same digest d signed by different replicas (including possibly its own such message). These $2f + 1$ messages are the proof of correctness for the checkpoint.

A checkpoint with a proof becomes stable and the replica discards all pre-prepare, prepare, and commit messages with sequence number less than or equal to n from its log; it also discards all earlier checkpoints and checkpoint messages.

Computing the proofs is efficient because the digest can be computed using incremental cryptography [1] as discussed in Section 6.3, and proofs are generated rarely.

The checkpoint protocol is used to advance the low and high water marks (which limit what messages will be accepted). The low-water mark h is equal to the sequence number of the last stable checkpoint. The high water mark $H = h + k$, where k is big enough so that replicas do not stall waiting for a checkpoint to become stable. For example, if checkpoints are taken every 100 requests, k might be 200.

4.4 View Changes

The view-change protocol provides liveness by allowing the system to make progress when the primary fails. View changes are triggered by timeouts that prevent backups from waiting indefinitely for requests to execute. A backup is *waiting* for a request if it received a valid request

and has not executed it. A backup starts a timer when it receives a request and the timer is not already running. It stops the timer when it is no longer waiting to execute the request, but restarts it if at that point it is waiting to execute some other request.

If the timer of backup i expires in view v, the backup starts a view change to move the system to view $v+1$. It stops accepting messages (other than checkpoint, view-change, and new-view messages) and multicasts a $\langle \text{VIEW-CHANGE}, v+1, n, \mathcal{C}, \mathcal{P}, i \rangle_{\sigma_i}$ message to all replicas. Here n is the sequence number of the last stable checkpoint s known to i, $\mathcal{C}$ is a set of $2f+1$ valid checkpoint messages proving the correctness of s, and $\mathcal{P}$ is a set containing a set $\mathcal{P}_m$ for each request m that prepared at i with a sequence number higher than n. Each set $\mathcal{P}_m$ contains a valid pre-prepare message (without the corresponding client message) and $2f$ matching, valid prepare messages signed by different backups with the same view, sequence number, and the digest of m.

When the primary p of view $v+1$ receives $2f$ valid view-change messages for view $v+1$ from other replicas, it multicasts a $\langle \text{NEW-VIEW}, v+1, \mathcal{V}, \mathcal{O} \rangle_{\sigma_p}$ message to all other replicas, where $\mathcal{V}$ is a set containing the valid view-change messages received by the primary plus the view-change message for $v+1$ the primary sent (or would have sent), and $\mathcal{O}$ is a set of pre-prepare messages (without the piggybacked request). $\mathcal{O}$ is computed as follows:

1. The primary determines the sequence number *min-s* of the latest stable checkpoint in $\mathcal{V}$ and the highest sequence number *max-s* in a prepare message in $\mathcal{V}$.
2. The primary creates a new pre-prepare message for view $v+1$ for each sequence number n between *min-s* and *max-s*. There are two cases: (1) there is at least one set in the $\mathcal{P}$ component of some view-change message in $\mathcal{V}$ with sequence number n, or (2) there is no such set. In the first case, the primary creates a new message $\langle \text{PRE-PREPARE}, v+1, n, d \rangle_{\sigma_p}$, where d is the request digest in the pre-prepare message for sequence number n with the highest view number in $\mathcal{V}$. In the second case, it creates a new pre-prepare message $\langle \text{PRE-PREPARE}, v+1, n, d^{null} \rangle_{\sigma_p}$, where d^{null} is the digest of a special *null* request; a null request goes through the protocol like other requests, but its execution is a no-op. (Paxos [18] used a similar technique to fill in gaps.)

Next the primary appends the messages in $\mathcal{O}$ to its log. If *min-s* is greater than the sequence number of its latest stable checkpoint, the primary also inserts the proof of stability for the checkpoint with sequence number *min-s* in its log, and discards information from the log as discussed in Section 4.3. Then it *enters* view $v+1$: at this point it is able to accept messages for view $v+1$.

A backup accepts a new-view message for view $v+1$ if it is signed properly, if the view-change messages it contains are valid for view $v+1$, and if the set $\mathcal{O}$ is correct; it verifies the correctness of $\mathcal{O}$ by performing a computation similar to the one used by the primary to create $\mathcal{O}$. Then it adds the new information to its log as described for the primary, multicasts a prepare for each message in $\mathcal{O}$ to all the other replicas, adds these prepares to its log, and enters view $v+1$.

Thereafter, the protocol proceeds as described in Section 4.2. Replicas redo the protocol for messages between *min-s* and *max-s* but they avoid re-executing client requests (by using their stored information about the last reply sent to each client).

A replica may be missing some request message m or a stable checkpoint (since these are not sent in new-view messages.) It can obtain missing information from another replica. For example, replica i can obtain a missing checkpoint state s from one of the replicas whose checkpoint messages certified its correctness in $\mathcal{V}$. Since $f+1$ of those replicas are correct, replica i will always obtain s or a later certified stable checkpoint. We can avoid sending the entire checkpoint by partitioning the state and stamping each partition with the sequence number of the last request that modified it. To bring a replica up to date, it is only necessary to send it the partitions where it is out of date, rather than the whole checkpoint.

4.5 Correctness

This section sketches the proof that the algorithm provides safety and liveness; details can be found in [4].

4.5.1 Safety

As discussed earlier, the algorithm provides safety if all non-faulty replicas agree on the sequence numbers of requests that commit locally.

In Section 4.2, we showed that if $prepared(m, v, n, i)$ is true, $prepared(m', v, n, j)$ is false for any non-faulty replica j (including $i = j$) and any m' such that $D(m') \neq D(m)$. This implies that two non-faulty replicas agree on the sequence number of requests that commit locally in the same view at the two replicas.

The view-change protocol ensures that non-faulty replicas also agree on the sequence number of requests that commit locally in different views at different replicas. A request m commits locally at a non-faulty replica with sequence number n in view v only if $committed(m, v, n)$ is true. This means that there is a set R_1 containing at least $f+1$ non-faulty replicas such that $prepared(m, v, n, i)$ is true for every replica i in the set.

Non-faulty replicas will not accept a pre-prepare for view $v' > v$ without having received a new-view message for v' (since only at that point do they enter the view). But any correct new-view message for view $v' > v$ contains correct view-change messages from every replica i in a

set R_2 of $2f + 1$ replicas. Since there are $3f + 1$ replicas, R_1 and R_2 must intersect in at least one replica k that is not faulty. k's view-change message will ensure that the fact that m prepared in a previous view is propagated to subsequent views, unless the new-view message contains a view-change message with a stable checkpoint with a sequence number higher than n. In the first case, the algorithm redoes the three phases of the atomic multicast protocol for m with the same sequence number n and the new view number. This is important because it prevents any different request that was assigned the sequence number n in a previous view from ever committing. In the second case no replica in the new view will accept any message with sequence number lower than n. In either case, the replicas will agree on the request that commits locally with sequence number n.

4.5.2 Liveness

To provide liveness, replicas must move to a new view if they are unable to execute a request. But it is important to maximize the period of time when at least $2f + 1$ non-faulty replicas are in the same view, and to ensure that this period of time increases exponentially until some requested operation executes. We achieve these goals by three means.

First, to avoid starting a view change too soon, a replica that multicasts a view-change message for view $v + 1$ waits for $2f + 1$ view-change messages for view $v + 1$ and then starts its timer to expire after some time T. If the timer expires before it receives a valid new-view message for $v + 1$ or before it executes a request in the new view that it had not executed previously, it starts the view change for view $v + 2$ but this time it will wait $2T$ before starting a view change for view $v + 3$.

Second, if a replica receives a set of $f + 1$ valid view-change messages from other replicas for views greater than its current view, it sends a view-change message for the smallest view in the set, even if its timer has not expired; this prevents it from starting the next view change too late.

Third, faulty replicas are unable to impede progress by forcing frequent view changes. A faulty replica cannot cause a view change by sending a view-change message, because a view change will happen only if at least $f + 1$ replicas send view-change messages, but it can cause a view change when it is the primary (by not sending messages or sending bad messages). However, because the primary of view v is the replica p such that $p = v \ mod \ |\mathcal{R}|$, the primary cannot be faulty for more than f consecutive views.

These three techniques guarantee liveness unless message delays grow faster than the timeout period indefinitely, which is unlikely in a real system.

4.6 Non-Determinism

State machine replicas must be deterministic but many services involve some form of non-determinism. For example, the time-last-modified in NFS is set by reading the server's local clock; if this were done independently at each replica, the states of non-faulty replicas would diverge. Therefore, some mechanism to ensure that all replicas select the same value is needed. In general, the client cannot select the value because it does not have enough information; for example, it does not know how its request will be ordered relative to concurrent requests by other clients. Instead, the primary needs to select the value either independently or based on values provided by the backups.

If the primary selects the non-deterministic value independently, it concatenates the value with the associated request and executes the three phase protocol to ensure that non-faulty replicas agree on a sequence number for the request and value. This prevents a faulty primary from causing replica state to diverge by sending different values to different replicas. However, a faulty primary might send the same, incorrect, value to all replicas. Therefore, replicas must be able to decide deterministicaly whether the value is correct (and what to do if it is not) based only on the service state.

This protocol is adequate for most services (including NFS) but occasionally replicas must participate in selecting the value to satisfy a service's specification. This can be accomplished by adding an extra phase to the protocol: the primary obtains authenticated values proposed by the backups, concatenates $2f + 1$ of them with the associated request, and starts the three phase protocol for the concatenated message. Replicas choose the value by a deterministic computation on the $2f + 1$ values and their state, e.g., taking the median. The extra phase can be optimized away in the common case. For example, if replicas need a value that is "close enough" to that of their local clock, the extra phase can be avoided when their clocks are synchronized within some delta.

5 Optimizations

This section describes some optimizations that improve the performance of the algorithm during normal-case operation. All the optimizations preserve the liveness and safety properties.

5.1 Reducing Communication

We use three optimizations to reduce the cost of communication. The first avoids sending most large replies. A client request designates a replica to send the result; all other replicas send replies containing just the digest of the result. The digests allow the client to check the correctness of the result while reducing network

bandwidth consumption and CPU overhead significantly for large replies. If the client does not receive a correct result from the designated replica, it retransmits the request as usual, requesting all replicas to send full replies.

The second optimization reduces the number of message delays for an operation invocation from 5 to 4. Replicas execute a request *tentatively* as soon as the prepared predicate holds for the request, their state reflects the execution of all requests with lower sequence number, and these requests are all known to have committed. After executing the request, the replicas send tentative replies to the client. The client waits for $2f + 1$ matching tentative replies. If it receives this many, the request is guaranteed to commit eventually. Otherwise, the client retransmits the request and waits for $f + 1$ non-tentative replies.

A request that has executed tentatively may abort if there is a view change and it is replaced by a *null* request. In this case the replica reverts its state to the last stable checkpoint in the new-view message or to its last checkpointed state (depending on which one has the higher sequence number).

The third optimization improves the performance of read-only operations that do not modify the service state. A client multicasts a read-only request to all replicas. Replicas execute the request immediately in their tentative state after checking that the request is properly authenticated, that the client has access, and that the request is in fact read-only. They send the reply only after all requests reflected in the tentative state have committed; this is necessary to prevent the client from observing uncommitted state. The client waits for $2f + 1$ replies from different replicas with the same result. The client may be unable to collect $2f + 1$ such replies if there are concurrent writes to data that affect the result; in this case, it retransmits the request as a regular read-write request after its retransmission timer expires.

5.2 Cryptography

In Section 4, we described an algorithm that uses digital signatures to authenticate all messages. However, we actually use digital signatures only for view-change and new-view messages, which are sent rarely, and authenticate all other messages using message authentication codes (MACs). This eliminates the main performance bottleneck in previous systems [29, 22].

However, MACs have a fundamental limitation relative to digital signatures — the inability to prove that a message is authentic to a third party. The algorithm in Section 4 and previous Byzantine-fault-tolerant algorithms [31, 16] for state machine replication rely on the extra power of digital signatures. We modified our algorithm to circumvent the problem by taking advantage of specific invariants, e.g, the invariant that no two different requests prepare with the same view and sequence number at two non-faulty replicas. The modified algorithm is described in [5]. Here we sketch the main implications of using MACs.

MACs can be computed three orders of magnitude faster than digital signatures. For example, a 200MHz Pentium Pro takes 43ms to generate a 1024-bit modulus RSA signature of an MD5 digest and 0.6ms to verify the signature [37], whereas it takes only 10.3μs to compute the MAC of a 64-byte message on the same hardware in our implementation. There are other public-key cryptosystems that generate signatures faster, e.g., elliptic curve public-key cryptosystems, but signature verification is slower [37] and in our algorithm each signature is verified many times.

Each node (including active clients) shares a 16-byte secret session key with each replica. We compute message authentication codes by applying MD5 to the concatenation of the message with the secret key. Rather than using the 16 bytes of the final MD5 digest, we use only the 10 least significant bytes. This truncation has the obvious advantage of reducing the size of MACs and it also improves their resilience to certain attacks [27]. This is a variant of the secret suffix method [36], which is secure as long as MD5 is collision resistant [27, 8].

The digital signature in a reply message is replaced by a single MAC, which is sufficient because these messages have a single intended recipient. The signatures in all other messages (including client requests but excluding view changes) are replaced by vectors of MACs that we call *authenticators*. An authenticator has an entry for every replica other than the sender; each entry is the MAC computed with the key shared by the sender and the replica corresponding to the entry.

The time to verify an authenticator is constant but the time to generate one grows linearly with the number of replicas. This is not a problem because we do not expect to have a large number of replicas and there is a huge performance gap between MAC and digital signature computation. Furthermore, we compute authenticators efficiently; MD5 is applied to the message once and the resulting context is used to compute each vector entry by applying MD5 to the corresponding session key. For example, in a system with 37 replicas (i.e., a system that can tolerate 12 simultaneous faults) an authenticator can still be computed much more than two orders of magnitude faster than a 1024-bit modulus RSA signature.

The size of authenticators grows linearly with the number of replicas but it grows slowly: it is equal to $30 \times \lfloor \frac{n-1}{3} \rfloor$ bytes. An authenticator is smaller than an RSA signature with a 1024-bit modulus for $n \leq 13$ (i.e., systems that can tolerate up to 4 simultaneous faults), which we expect to be true in most configurations.

6 Implementation

This section describes our implementation. First we discuss the replication library, which can be used as a basis for any replicated service. In Section 6.2 we describe how we implemented a replicated NFS on top of the replication library. Then we describe how we maintain checkpoints and compute checkpoint digests efficiently.

6.1 The Replication Library

The client interface to the replication library consists of a single procedure, *invoke*, with one argument, an input buffer containing a request to invoke a state machine operation. The *invoke* procedure uses our protocol to execute the requested operation at the replicas and select the correct reply from among the replies of the individual replicas. It returns a pointer to a buffer containing the operation result.

On the server side, the replication code makes a number of upcalls to procedures that the server part of the application must implement. There are procedures to execute requests (*execute*), to maintain checkpoints of the service state (*make_checkpoint*, *delete_checkpoint*), to obtain the digest of a specified checkpoint (*get_digest*), and to obtain missing information (*get_checkpoint*, *set_checkpoint*). The *execute* procedure receives as input a buffer containing the requested operation, executes the operation, and places the result in an output buffer. The other procedures are discussed further in Sections 6.3 and 6.4.

Point-to-point communication between nodes is implemented using UDP, and multicast to the group of replicas is implemented using UDP over IP multicast [7]. There is a single IP multicast group for each service, which contains all the replicas. These communication protocols are unreliable; they may duplicate or lose messages or deliver them out of order.

The algorithm tolerates out-of-order delivery and rejects duplicates. View changes can be used to recover from lost messages, but this is expensive and therefore it is important to perform retransmissions. During normal operation recovery from lost messages is driven by the receiver: backups send negative acknowledgments to the primary when they are out of date and the primary retransmits pre-prepare messages after a long timeout. A reply to a negative acknowledgment may include both a portion of a stable checkpoint and missing messages. During view changes, replicas retransmit view-change messages until they receive a matching new-view message or they move on to a later view.

The replication library does not implement view changes or retransmissions at present. This does not compromise the accuracy of the results given in Section 7 because the rest of the algorithm is completely implemented (including the manipulation of the timers that trigger view changes) and because we have formalized the complete algorithm and proved its correctness [4].

6.2 BFS: A Byzantine-Fault-tolerant File System

We implemented BFS, a Byzantine-fault-tolerant NFS service, using the replication library. Figure 2 shows the architecture of BFS. We opted not to modify the kernel NFS client and server because we did not have the sources for the Digital Unix kernel.

A file system exported by the fault-tolerant NFS service is mounted on the client machine like any regular NFS file system. Application processes run unmodified and interact with the mounted file system through the NFS client in the kernel. We rely on user level *relay* processes to mediate communication between the standard NFS client and the replicas. A relay receives NFS protocol requests, calls the *invoke* procedure of our replication library, and sends the result back to the NFS client.

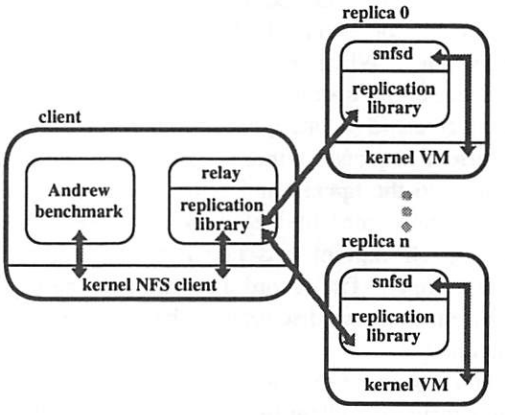


Figure 2: Replicated File System Architecture.

Each replica runs a user-level process with the replication library and our NFS V2 daemon, which we will refer to as *snfsd* (for simple *nfsd*). The replication library receives requests from the relay, interacts with *snfsd* by making upcalls, and packages NFS replies into replication protocol replies that it sends to the relay.

We implemented *snfsd* using a fixed-size memory-mapped file. All the file system data structures, e.g., inodes, blocks and their free lists, are in the mapped file. We rely on the operating system to manage the cache of memory-mapped file pages and to write modified pages to disk asynchronously. The current implementation uses 8KB blocks and inodes contain the NFS status information plus 256 bytes of data, which is used to store directory entries in directories, pointers to blocks in files, and text in symbolic links. Directories and files may also use indirect blocks in a way similar to Unix.

Our implementation ensures that all state machine

replicas start in the same initial state and are deterministic, which are necessary conditions for the correctness of a service implemented using our protocol. The primary proposes the values for time-last-modified and time-last-accessed, and replicas select the larger of the proposed value and one greater than the maximum of all values selected for earlier requests. We do not require synchronous writes to implement NFS V2 protocol semantics because BFS achieves stability of modified data and meta-data through replication [20].

6.3 Maintaining Checkpoints

This section describes how *snfsd* maintains checkpoints of the file system state. Recall that each replica maintains several logical copies of the state: the current state, some number of checkpoints that are not yet stable, and the last stable checkpoint.

snfsd executes file system operations directly in the memory mapped file to preserve locality, and it uses copy-on-write to reduce the space and time overhead associated with maintaining checkpoints. *snfsd* maintains a copy-on-write bit for every 512-byte block in the memory mapped file. When the replication code invokes the *make_checkpoint* upcall, *snfsd* sets all the copy-on-write bits and creates a (volatile) checkpoint record, containing the current sequence number, which it receives as an argument to the upcall, and a list of blocks. This list contains the copies of the blocks that were modified since the checkpoint was taken, and therefore, it is initially empty. The record also contains the digest of the current state; we discuss how the digest is computed in Section 6.4.

When a block of the memory mapped file is modified while executing a client request, *snfsd* checks the copy-on-write bit for the block and, if it is set, stores the block's current contents and its identifier in the checkpoint record for the last checkpoint. Then, it overwrites the block with its new value and resets its copy-on-write bit. *snfsd* retains a checkpoint record until told to discard it via a *delete_checkpoint* upcall, which is made by the replication code when a later checkpoint becomes stable.

If the replication code requires a checkpoint to send to another replica, it calls the *get_checkpoint* upcall. To obtain the value for a block, *snfsd* first searches for the block in the checkpoint record of the stable checkpoint, and then searches the checkpoint records of any later checkpoints. If the block is not in any checkpoint record, it returns the value from the current state.

The use of the copy-on-write technique and the fact that we keep at most 2 checkpoints ensure that the space and time overheads of keeping several logical copies of the state are low. For example, in the Andrew benchmark experiments described in Section 7, the average checkpoint record size is only 182 blocks with a maximum of 500.

6.4 Computing Checkpoint Digests

snfsd computes a digest of a checkpoint state as part of a *make_checkpoint* upcall. Although checkpoints are only taken occasionally, it is important to compute the state digest incrementally because the state may be large. *snfsd* uses an incremental collision-resistant one-way hash function called AdHash [1]. This function divides the state into fixed-size blocks and uses some other hash function (e.g., MD5) to compute the digest of the string obtained by concatenating the block index with the block value for each block. The digest of the state is the sum of the digests of the blocks modulo some large integer. In our current implementation, we use the 512-byte blocks from the copy-on-write technique and compute their digest using MD5.

To compute the digest for the state incrementally, *snfsd* maintains a table with a hash value for each 512-byte block. This hash value is obtained by applying MD5 to the block index concatenated with the block value at the time of the last checkpoint. When *make_checkpoint* is called, *snfsd* obtains the digest d for the previous checkpoint state (from the associated checkpoint record). It computes new hash values for each block whose copy-on-write bit is reset by applying MD5 to the block index concatenated with the current block value. Then, it adds the new hash value to d, subtracts the old hash value from d, and updates the table to contain the new hash value. This process is efficient provided the number of modified blocks is small; as mentioned above, on average 182 blocks are modified per checkpoint for the Andrew benchmark.

7 Performance Evaluation

This section evaluates the performance of our system using two benchmarks: a micro-benchmark and the Andrew benchmark [15]. The micro-benchmark provides a service-independent evaluation of the performance of the replication library; it measures the latency to invoke a null operation, i.e., an operation that does nothing.

The Andrew benchmark is used to compare BFS with two other file systems: one is the NFS V2 implementation in Digital Unix, and the other is identical to BFS except without replication. The first comparison demonstrates that our system is practical by showing that its latency is similar to the latency of a commercial system that is used daily by many users. The second comparison allows us to evaluate the overhead of our algorithm accurately within an implementation of a real service.

7.1 Experimental Setup

The experiments measure normal-case behavior (i.e., there are no view changes), because this is the behavior

that determines the performance of the system. All experiments ran with one client running two relay processes, and four replicas. Four replicas can tolerate one Byzantine fault; we expect this reliability level to suffice for most applications. The replicas and the client ran on identical DEC 3000/400 Alpha workstations. These workstations have a 133 MHz Alpha 21064 processor, 128 MB of memory, and run Digital Unix version 4.0. The file system was stored by each replica on a DEC RZ26 disk. All the workstations were connected by a 10Mbit/s switched Ethernet and had DEC LANCE Ethernet interfaces. The switch was a DEC EtherWORKS 8T/TX. The experiments were run on an isolated network.

The interval between checkpoints was 128 requests, which causes garbage collection to occur several times in any of the experiments. The maximum sequence number accepted by replicas in pre-prepare messages was 256 plus the sequence number of the last stable checkpoint.

7.2 Micro-Benchmark

The micro-benchmark measures the latency to invoke a null operation. It evaluates the performance of two implementations of a simple service with no state that implements null operations with arguments and results of different sizes. The first implementation is replicated using our library and the second is unreplicated and uses UDP directly. Table 1 reports the response times measured at the client for both read-only and read-write operations. They were obtained by timing 10,000 operation invocations in three separate runs and we report the median value of the three runs. The maximum deviation from the median was always below 0.3% of the reported value. We denote each operation by *a/b*, where *a* and *b* are the sizes of the operation argument and result in KBytes.

arg./res. (KB)	replicated		without replication
	read-write	read-only	
0/0	3.35 (309%)	1.62 (98%)	0.82
4/0	14.19 (207%)	6.98 (51%)	4.62
0/4	8.01 (72%)	5.94 (27%)	4.66

Table 1: Micro-benchmark results (in milliseconds); the percentage overhead is relative to the unreplicated case.

The overhead introduced by the replication library is due to extra computation and communication. For example, the computation overhead for the read-write 0/0 operation is approximately 1.06ms, which includes 0.55ms spent executing cryptographic operations. The remaining 1.47ms of overhead are due to extra communication; the replication library introduces an extra message round-trip, it sends larger messages, and it increases the number of messages received by each node relative to the service without replication.

The overhead for read-only operations is significantly lower because the optimization discussed in Section 5.1 reduces both computation and communication overheads. For example, the computation overhead for the read-only 0/0 operation is approximately 0.43ms, which includes 0.23ms spent executing cryptographic operations, and the communication overhead is only 0.37ms because the protocol to execute read-only operations uses a single round-trip.

Table 1 shows that the relative overhead is lower for the 4/0 and 0/4 operations. This is because a significant fraction of the overhead introduced by the replication library is independent of the size of operation arguments and results. For example, in the read-write 0/4 operation, the large message (the reply) goes over the network only once (as discussed in Section 5.1) and only the cryptographic overhead to process the reply message is increased. The overhead is higher for the read-write 4/0 operation because the large message (the request) goes over the network twice and increases the cryptographic overhead for processing both request and pre-prepare messages.

It is important to note that this micro-benchmark represents the worst case overhead for our algorithm because the operations perform no work and the unreplicated server provides very weak guarantees. Most services will require stronger guarantees, e.g., authenticated connections, and the overhead introduced by our algorithm relative to a server that implements these guarantees will be lower. For example, the overhead of the replication library relative to a version of the unreplicated service that uses MACs for authentication is only 243% for the read-write 0/0 operation and 4% for the read-only 4/0 operation.

We can estimate a rough lower bound on the performance gain afforded by our algorithm relative to Rampart [30]. Reiter reports that Rampart has a latency of 45ms for a multi-RPC of a null message in a 10 Mbit/s Ethernet network of 4 SparcStation 10s [30]. The multi-RPC is sufficient for the primary to invoke a state machine operation but for an arbitrary client to invoke an operation it would be necessary to add an extra message delay and an extra RSA signature and verification to authenticate the client; this would lead to a latency of at least 65ms (using the RSA timings reported in [29].) Even if we divide this latency by 1.7, the ratio of the SPECint92 ratings of the DEC 3000/400 and the SparcStation 10, our algorithm still reduces the latency to invoke the read-write and read-only 0/0 operations by factors of more than 10 and 20, respectively. Note that this scaling is conservative because the network accounts for a significant fraction of Rampart's latency [29] and Rampart's results were obtained using 300-bit modulus RSA signatures, which are not considered secure today unless the keys used to

generate them are refreshed very frequently.

There are no published performance numbers for SecureRing [16] but it would be slower than Rampart because its algorithm has more message delays and signature operations in the critical path.

7.3 Andrew Benchmark

The Andrew benchmark [15] emulates a software development workload. It has five phases: (1) creates subdirectories recursively; (2) copies a source tree; (3) examines the status of all the files in the tree without examining their data; (4) examines every byte of data in all the files; and (5) compiles and links the files.

We use the Andrew benchmark to compare BFS with two other file system configurations: NFS-std, which is the NFS V2 implementation in Digital Unix, and BFS-nr, which is identical to BFS but with no replication. BFS-nr ran two simple UDP relays on the client, and on the server it ran a thin veneer linked with a version of *snfsd* from which all the checkpoint management code was removed. This configuration does *not* write modified file system state to disk before replying to the client. Therefore, it does not implement NFS V2 protocol semantics, whereas both BFS and NFS-std do.

Out of the 18 operations in the NFS V2 protocol only `getattr` is read-only because the time-last-accessed attribute of files and directories is set by operations that would otherwise be read-only, e.g., `read` and `lookup`. The result is that our optimization for read-only operations can rarely be used. To show the impact of this optimization, we also ran the Andrew benchmark on a second version of BFS that modifies the `lookup` operation to be read-only. This modification violates strict Unix file system semantics but is unlikely to have adverse effects in practice.

For all configurations, the actual benchmark code ran at the client workstation using the standard NFS client implementation in the Digital Unix kernel with the same `mount` options. The most relevant of these options for the benchmark are: UDP transport, 4096-byte read and write buffers, allowing asynchronous client writes, and allowing attribute caching.

We report the mean of 10 runs of the benchmark for each configuration. The sample standard deviation for the total time to run the benchmark was always below 2.6% of the reported value but it was as high as 14% for the individual times of the first four phases. This high variance was also present in the NFS-std configuration. The estimated error for the reported mean was below 4.5% for the individual phases and 0.8% for the total.

Table 2 shows the results for BFS and BFS-nr. The comparison between BFS-strict and BFS-nr shows that the overhead of Byzantine fault tolerance for this service is low — BFS-strict takes only 26% more time to run the complete benchmark. The overhead is lower than what was observed for the micro-benchmarks because the client spends a significant fraction of the elapsed time computing between operations, i.e., between receiving the reply to an operation and issuing the next request, and operations at the server perform some computation. But the overhead is not uniform across the benchmark phases. The main reason for this is a variation in the amount of time the client spends computing between operations; the first two phases have a higher relative overhead because the client spends approximately 40% of the total time computing between operations, whereas it spends approximately 70% during the last three phases.

	BFS		
phase	strict	r/o lookup	BFS-nr
1	0.55 (57%)	0.47 (34%)	0.35
2	9.24 (82%)	7.91 (56%)	5.08
3	7.24 (18%)	6.45 (6%)	6.11
4	8.77 (18%)	7.87 (6%)	7.41
5	38.68 (20%)	38.38 (19%)	32.12
total	64.48 (26%)	61.07 (20%)	51.07

Table 2: Andrew benchmark: BFS vs BFS-nr. The times are in seconds.

The table shows that applying the read-only optimization to `lookup` improves the performance of BFS significantly and reduces the overhead relative to BFS-nr to 20%. This optimization has a significant impact in the first four phases because the time spent waiting for `lookup` operations to complete in BFS-strict is at least 20% of the elapsed time for these phases, whereas it is less than 5% of the elapsed time for the last phase.

	BFS		
phase	strict	r/o lookup	NFS-std
1	0.55 (-69%)	0.47 (-73%)	1.75
2	9.24 (-2%)	7.91 (-16%)	9.46
3	7.24 (35%)	6.45 (20%)	5.36
4	8.77 (32%)	7.87 (19%)	6.60
5	38.68 (-2%)	38.38 (-2%)	39.35
total	64.48 (3%)	61.07 (-2%)	62.52

Table 3: Andrew benchmark: BFS vs NFS-std. The times are in seconds.

Table 3 shows the results for BFS vs NFS-std. These results show that BFS can be used in practice — BFS-strict takes only 3% more time to run the complete benchmark. Thus, one could replace the NFS V2 implementation in Digital Unix, which is used daily by many users, by BFS without affecting the latency perceived by those users. Furthermore, BFS with the read-only optimization for the `lookup` operation is actually 2% faster than NFS-std.

The overhead of BFS relative to NFS-std is not the

same for all phases. Both versions of BFS are faster than NFS-std for phases 1, 2, and 5 but slower for the other phases. This is because during phases 1, 2, and 5 a large fraction (between 21% and 40%) of the operations issued by the client are *synchronous*, i.e., operations that require the NFS implementation to ensure stability of modified file system state before replying to the client. NFS-std achieves stability by writing modified state to disk whereas BFS achieves stability with lower latency using replication (as in Harp [20]). NFS-std is faster than BFS (and BFS-nr) in phases 3 and 4 because the client issues no synchronous operations during these phases.

8 Related Work

Most previous work on replication techniques ignored Byzantine faults or assumed a synchronous system model (e.g., [17, 26, 18, 34, 6, 10]). Viewstamped replication [26] and Paxos [18] use views with a primary and backups to tolerate benign faults in an asynchronous system. Tolerating Byzantine faults requires a much more complex protocol with cryptographic authentication, an extra pre-prepare phase, and a different technique to trigger view changes and select primaries. Furthermore, our system uses view changes only to select a new primary but never to select a different set of replicas to form the new view as in [26, 18].

Some agreement and consensus algorithms tolerate Byzantine faults in asynchronous systems (e.g,[2, 3, 24]). However, they do not provide a complete solution for state machine replication, and furthermore, most of them were designed to demonstrate theoretical feasibility and are too slow to be used in practice. Our algorithm during normal-case operation is similar to the Byzantine agreement algorithm in [2] but that algorithm is unable to survive primary failures.

The two systems that are most closely related to our work are Rampart [29, 30, 31, 22] and SecureRing [16]. They implement state machine replication but are more than an order of magnitude slower than our system and, most importantly, they rely on synchrony assumptions.

Both Rampart and SecureRing must exclude faulty replicas from the group to make progress (e.g., to remove a faulty primary and elect a new one), and to perform garbage collection. They rely on failure detectors to determine which replicas are faulty. However, failure detectors cannot be accurate in an asynchronous system [21], i.e., they may misclassify a replica as faulty. Since correctness requires that fewer than $1/3$ of group members be faulty, a misclassification can compromise correctness by removing a non-faulty replica from the group. This opens an avenue of attack: an attacker gains control over a single replica but does not change its behavior in any detectable way; then it slows correct replicas or the communication between them until enough are excluded from the group.

To reduce the probability of misclassification, failure detectors can be calibrated to delay classifying a replica as faulty. However, for the probability to be negligible the delay must be very large, which is undesirable. For example, if the primary has actually failed, the group will be unable to process client requests until the delay has expired. Our algorithm is not vulnerable to this problem because it never needs to exclude replicas from the group.

Phalanx [23, 25] applies quorum replication techniques [12] to achieve Byzantine fault-tolerance in asynchronous systems. This work does not provide generic state machine replication; instead, it offers a data repository with operations to read and write individual variables and to acquire locks. The semantics it provides for read and write operations are weaker than those offered by our algorithm; we can implement arbitrary operations that access any number of variables, whereas in Phalanx it would be necessary to acquire and release locks to execute such operations. There are no published performance numbers for Phalanx but we believe our algorithm is faster because it has fewer message delays in the critical path and because of our use of MACs rather than public key cryptography. The approach in Phalanx offers the potential for improved scalability; each operation is processed by only a subset of replicas. But this approach to scalability is expensive: it requires $n > 4f + 1$ to tolerate f faults; each replica needs a copy of the state; and the load on each replica decreases slowly with n (it is $O(1/\sqrt{n})$).

9 Conclusions

This paper has described a new state-machine replication algorithm that is able to tolerate Byzantine faults and can be used in practice: it is the first to work correctly in an asynchronous system like the Internet and it improves the performance of previous algorithms by more than an order of magnitude.

The paper also described BFS, a Byzantine-fault-tolerant implementation of NFS. BFS demonstrates that it is possible to use our algorithm to implement real services with performance close to that of an unreplicated service — the performance of BFS is only 3% worse than that of the standard NFS implementation in Digital Unix. This good performance is due to a number of important optimizations, including replacing public-key signatures by vectors of message authentication codes, reducing the size and number of messages, and the incremental checkpoint-management techniques.

One reason why Byzantine-fault-tolerant algorithms will be important in the future is that they can allow systems to continue to work correctly even when there are software errors. Not all errors are survivable; our approach cannot mask a software error that occurs

at all replicas. However, it can mask errors that occur independently at different replicas, including nondeterministic software errors, which are the most problematic and persistent errors since they are the hardest to detect. In fact, we encountered such a software bug while running our system, and our algorithm was able to continue running correctly in spite of it.

There is still much work to do on improving our system. One problem of special interest is reducing the amount of resources required to implement our algorithm. The number of replicas can be reduced by using f replicas as witnesses that are involved in the protocol only when some full replica fails. We also believe that it is possible to reduce the number of copies of the state to $f + 1$ but the details remain to be worked out.

Acknowledgments

We would like to thank Atul Adya, Chandrasekhar Boyapati, Nancy Lynch, Sape Mullender, Andrew Myers, Liuba Shrira, and the anonymous referees for their helpful comments on drafts of this paper.

References

[1] M. Bellare and D. Micciancio. A New Paradigm for Collision-free Hashing: Incrementality at Reduced Cost. In *Advances in Cryptology – Eurocrypt 97*, 1997.

[2] G. Bracha and S. Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4), 1995.

[3] R. Canneti and T. Rabin. Optimal Asynchronous Byzantine Agreement. Technical Report #92-15, Computer Science Department, Hebrew University, 1992.

[4] M. Castro and B. Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.

[5] M. Castro and B. Liskov. Authenticated Byzantine Fault Tolerance Without Public-Key Cryptography. Technical Memo MIT/LCS/TM-589, MIT Laboratory for Computer Science, 1999.

[6] F. Cristian, H. Aghili, H. Strong, and D. Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *International Conference on Fault Tolerant Computing*, 1985.

[7] S. Deering and D. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 8(2), 1990.

[8] H. Dobbertin. The Status of MD5 After a Recent Attack. *RSA Laboratories' CryptoBytes*, 2(2), 1996.

[9] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus With One Faulty Process. *Journal of the ACM*, 32(2), 1985.

[10] J. Garay and Y. Moses. Fully Polynomial Byzantine Agreement for n >3t Processors in t+1 Rounds. *SIAM Journal of Computing*, 27(1), 1998.

[11] D. Gawlick and D. Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *Database Engineering*, 8(2), 1985.

[12] D. Gifford. Weighted Voting for Replicated Data. In *Symposium on Operating Systems Principles*, 1979.

[13] M. Herlihy and J. Tygar. How to make replicated data secure. *Advances in Cryptology (LNCS 293)*, 1988.

[14] M. Herlihy and J. Wing. Axioms for Concurrent Objects. In *ACM Symposium on Principles of Programming Languages*, 1987.

[15] J. Howard et al. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), 1988.

[16] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Hawaii International Conference on System Sciences*, 1998.

[17] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7), 1978.

[18] L. Lamport. The Part-Time Parliament. Technical Report 49, DEC Systems Research Center, 1989.

[19] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

[20] B. Liskov et al. Replication in the Harp File System. In *ACM Symposium on Operating System Principles*, 1991.

[21] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[22] D. Malkhi and M. Reiter. A High-Throughput Secure Reliable Multicast Protocol. In *Computer Security Foundations Workshop*, 1996.

[23] D. Malkhi and M. Reiter. Byzantine Quorum Systems. In *ACM Symposium on Theory of Computing*, 1997.

[24] D. Malkhi and M. Reiter. Unreliable Intrusion Detection in Distributed Computations. In *Computer Security Foundations Workshop*, 1997.

[25] D. Malkhi and M. Reiter. Secure and Scalable Replication in Phalanx. In *IEEE Symposium on Reliable Distributed Systems*, 1998.

[26] B. Oki and B. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *ACM Symposium on Principles of Distributed Computing*, 1988.

[27] B. Preneel and P. Oorschot. MDx-MAC and Building Fast MACs from Hash Functions. In *Crypto 95*, 1995.

[28] C. Pu, A. Black, C. Cowan, and J. Walpole. A Specialization Toolkit to Increase the Diversity of Operating Systems. In *ICMAS Workshop on Immunity-Based Systems*, 1996.

[29] M. Reiter. Secure Agreement Protocols. In *ACM Conference on Computer and Communication Security*, 1994.

[30] M. Reiter. The Rampart Toolkit for Building High-Integrity Services. *Theory and Practice in Distributed Systems (LNCS 938)*, 1995.

[31] M. Reiter. A Secure Group Membership Protocol. *IEEE Transactions on Software Engineering*, 22(1), 1996.

[32] R. Rivest. The MD5 Message-Digest Algorithm. Internet RFC-1321, 1992.

[33] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2), 1978.

[34] F. Schneider. Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4), 1990.

[35] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11), 1979.

[36] G. Tsudik. Message Authentication with One-Way Hash Functions. *ACM Computer Communications Review*, 22(5), 1992.

[37] M. Wiener. Performance Comparison of Public-Key Cryptosystems. *RSA Laboratories' CryptoBytes*, 4(1), 1998.

The Coign Automatic Distributed Partitioning System

Galen C. Hunt
Microsoft Research
One Microsoft Way
Redmond, WA 98052
galenh@microsoft.com

Michael L. Scott
Department of Computer Science
University of Rochester
Rochester, NY 14627
scott@cs.rochester.edu

Abstract

Although successive generations of middleware (such as RPC, CORBA, and DCOM) have made it easier to connect distributed programs, the process of distributed application decomposition has changed little: programmers manually divide applications into sub-programs and manually assign those sub-programs to machines. Often the techniques used to choose a distribution are ad hoc and create one-time solutions biased to a specific combination of users, machines, and networks.

We assert that system software, not the programmer, should manage the task of distributed decomposition. To validate our assertion we present Coign, an automatic distributed partitioning system that significantly eases the development of distributed applications.

Given an application (in binary form) built from distributable COM components, Coign constructs a graph model of the application's inter-component communication through scenario-based profiling. Later, Coign applies a graph-cutting algorithm to partition the application across a network and minimize execution delay due to network communication. Using Coign, even an end user (without access to source code) can transform a non-distributed application into an optimized, distributed application.

Coign has automatically distributed binaries from over 2 million lines of application code, including Microsoft's PhotoDraw 2000 image processor. To our knowledge, Coign is the first system to automatically partition and distribute binary applications.

1. Introduction

Distributed systems have been an area of open research for more than two decades. Popular acceptance of the Internet has fueled a renewed interest in distributed systems and applications. Distributed application enable sharing of data, sharing of resources (such as memory, processor cycles, or physical devices), collaboration between users, increased reliability through redundancy, and increased security through physical isolation.

However compelling the motivations, the creation of distributed applications continues to be difficult. As a rule, the creation of a distributed application is always harder than the creation of a functionally equivalent non-distributed application. Complicating factors include protection of data integrity and security, management of disjoint address spaces, increased latency and reduced bandwidth between application components, partial system failures caused by isolated machine or network outages, and practical engineering issues such as debugging across multiple processes on distributed computers.

One of the primary challenges to create a distributed application is the need to partition and place pieces of the application. Although successive generations of middleware (such as RPC [4, 15, 34], CORBA [35, 42], and DCOM [8]) have brought the advantages of service-location transparency, dynamic object instantiation, and object-oriented programming to distributed applications, the process of distributed application decomposition has changed little: programmers manually divide applications into sub-programs and manually assign those sub-programs to machines. Often the techniques used to choose a distribution are ad hoc, creating solutions biased to a specific platform.

Given the effort required, applications are seldom repartitioned even in drastically different network environments. Changes in underlying network, from ISDN to 100BaseT to ATM to SAN, strain static distributions as bandwidth-to-latency tradeoffs change by more than an order of magnitude. User usage patterns can also severely stress a static application distribution. Nevertheless, programmers resist repartitioning applications because doing so often requires extensive modifications to program structure and source code.

We argue that system software, not the programmer, should partition and distribute applications. Furthermore, we assert that existing applications can be partitioned and distributed automatically without access to source code, provided the applications are built from binary components. To validate our claims, we have built a working prototype system, the Coign *Automatic Distributed Partitioning System* (ADPS).

Coign radically changes the development of distributed applications. Given an application built with components conforming to Microsoft's Component Object Model (COM), Coign profiles inter-component communication as the application is run through typical usage scenarios (a process known as *scenario-based profiling*). Based on profiled information, Coign selects a distribution of the application with minimal communication time for a particular distributed environment. Coign then modifies the application to produce the desired distribution.

Coign analyzes an application, chooses a distribution, and produces the desired distributed application all with access to only the application binary files. By solely analyzing application binaries, Coign produces distributed applications automatically without violating the primary goal of commercial component systems: building applications from reusable, binary components.

In the next two sections, we describe Coign and the implementation of the Coign runtime. In Section 4, we present experimental results demonstrating Coign's effectiveness in automatically distributing binaries from over 2 million lines of application code. In Section 5, we describe related work. Finally, in Section 6 we summarize our conclusions and discuss future work.

2. System Description

Coign is an automatic distributed partitioning system (ADPS) for applications built from COM components. COM is a standard for packaging, instantiating, and connecting reusable pieces of software in binary form called components. Clients talk to components through polymorphic interfaces. Abstractly, a COM interface is a collection of semantically related function entry points. Concretely, COM defines a binary standard representation of an interface as a virtual function table. All first-class communication between COM components passes through interfaces. Clients reference a component through pointers to its interfaces.

Due to a strict binary standard, COM can transparently interpose proxies, stubs, and middleware layers between communicating clients and components for true location transparency. Application code remains identical for in-process, cross-process, and cross-machine communication. The DCOM protocol, a superset of DCE RPC [18], transports messages between machines by deep copy of message arguments. By leveraging the COM binary standard, Coign can automatically distribute an application without any knowledge of the application source code.

The Coign ADPS consists of four major tools: the Coign run-time, a binary rewriter, a network profiler, and a profile analysis engine. Figure 1 contains an overview of the Coign ADPS.

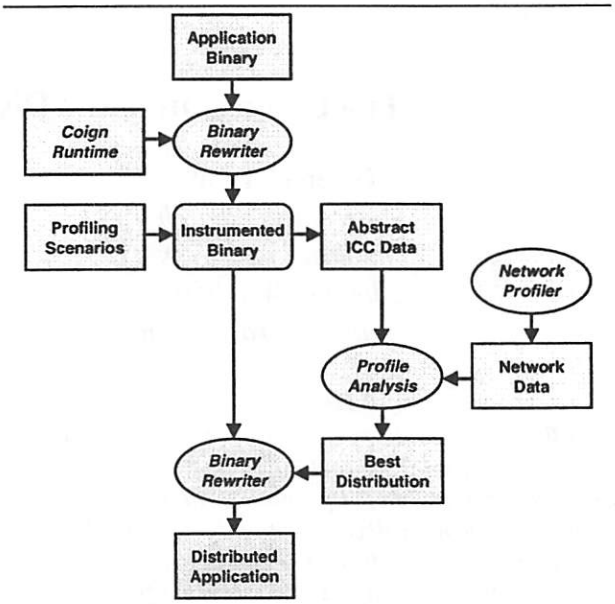


Figure 1. The Coign ADPS.
An application is transformed into a distributed application by inserting the Coign runtime, profiling the instrumented application, and analyzing the profiles to cut the network-based ICC graph.

Starting with the original binary files for an application, the *binary rewriter* creates a Coign-instrumented version of the application. The binary rewriter makes two modifications to the application. First, it inserts an entry into the first slot of the application's dynamic link library (DLL) import table to load the Coign runtime. Second, it adds a data segment containing configuration information at the end of application binary. The configuration information tells the Coign runtime how to profile the application and how to classify components during execution.

Because it occupies the first slot in the application's DLL import table, the Coign runtime always loads and executes before the application or any of its DLLs. At load time, the Coign runtime inserts binary instrumentation into the images of system libraries in the application's address space. The instrumentation traps all component instantiation requests in the COM library.

The instrumented binary is run through a set of profiling scenarios. Because the binary modifications are transparent to the user (and to the application itself), the instrumented binary behaves identically to the original application. The instrumentation gathers profiling information in the background while the user controls the application. The only visible effect of profiling is a small degradation in application performance (of up to 85%). For advanced profiling, scenarios can be driven by an automated testing tool, such as Visual Test [2].

During profiling, the Coign instrumentation summarizes inter-component communication within the appli-

cation. Every inter-component call is executed via a COM interface. Coign intercepts these interface calls (by instrumenting all component interfaces) and measures the amount of data communicated. The instrumentation measures the number of bytes that would be transferred from one machine to another *if* the two communicating components were distributed. It does so by invoking portions of the DCOM code, including interface proxies and stubs, within the application's address space. Coign measurement follows precisely the deep-copy semantics of DCOM. After quantifying communication (by number and size of messages), Coign compresses and summarizes the data online. Consequently, the overhead for storing communication information does not grow linearly with execution time. If desired, the application may be run through profiling scenarios for days or even weeks to more accurately track user usage patterns.

At the end of a profiling execution, Coign writes the inter-component communication profiles to a file for later analysis. In addition to information about the number and size of messages and components in the application, the profile log also contains information to classify components and to determine component location constraints. Log files from multiple profiling scenarios may be combined and summarized during later analysis. Alternatively, at the end of each profiling scenario, information from the log file may be combined into the configuration record in the application binary. The latter approach uses less storage because summary information in the configuration record accumulates communication from similar interface calls into a single entry.

The *profile analysis engine* combines component communication profiles and component location constraints to create an abstract inter-component communication (ICC) graph of the application. Location constraints can be acquired from the programmer, from analysis of component communication records, and from application binaries. For client-server distributions, the analysis engine performs static analysis on component binaries to determine which Windows APIs are called by each component. Components that access a set of known GUI or storage APIs are placed on the client or server respectively. Other components are distributed based on communication analysis.

The abstract ICC graph is combined with a network profile to create a concrete graph of potential communication time on the network. The *network profiler* creates a network profile through statistically sampling of communication time for a representative set of DCOM messages.

Coign employs the *lift-to-front minimum-cut* graph-cutting algorithm [9] to choose a distribution with minimal communication time. In the future, the concrete graph could be constructed and cut at application execution time, thus introducing the potential to produce a new distribution tailored to current network characteristics for each execution.

The lift-to-front min-cut algorithm, in our current implementation, can produce only two-machine, client-server applications. The problem of partitioning applications across three or more machines is provably NP-hard [13]. Numerous heuristic algorithms exist for multi-way graph cutting [7, 10, 12, 33, 38]. To more accurately evaluate the rest of the system, we restrict ourselves to an exact, two-way algorithm for client-server computing.

After analysis, the application's ICC graph and component classification data (to be described later) are written into the configuration record in the application binary. The configuration record is also modified to remove the profiling instrumentation. In its place, a lightweight version of the instrumentation will be loaded to realize (enforce) the distribution chosen by the graph-cutting algorithm.

3. Coign Runtime Description

The Coign runtime is composed of a small collection of replaceable COM components (Figure 2). The most important components are the *Coign Runtime Executive* (RTE), the *interface informer*, the *information logger*, the *instance classifier*, and the *component factory*. The RTE provides low-level services to other components in the Coign runtime. The *interface informer* walks the parameters of interface function calls and identifies the location and static type of component interfaces. The *information logger* records data necessary for post-profiling analysis. The *instance classifier* identifies component instances with similar communication profiles across multiple program executions. The *component factory* decides where component instantiation requests should be fulfilled and relocates instantiation as needed to produce a chosen distribution. The component structure of the Coign runtime facilitates its use for a wide variety of application analysis and adaptation.

3.1. Runtime Executive.

The Coign Runtime Executive (RTE) provides low-level services to other components in the Coign runtime. Services provided by the RTE include:

Interception of component instantiation requests. The RTE traps all component instantiation requests made by the application to the COM runtime. Instantiation requests are trapped using inline redirection (similar to the techniques pioneered by the Parasight [1]

parallel debugger)[1]. The RTE invokes the instance classifier to identify the about-to-be-instantiated component. The RTE then invokes the component factory, which fulfills the instantiation request at the appropriate location based on instance classification.

Interface wrapping. The RTE "wraps" all COM interfaces by replacing each component interface pointer with a pointer to a Coign instrumented interface, which in turn forwards incoming calls through the original interface pointer. Once an interface is wrapped, the Coign runtime can trap all calls across the interface. An interface is wrapped using static information from the interface informer. The RTE also invokes the interface informer to process the parameters of interface function calls.

Address space and private stack management. The RTE tracks all binaries (.DLL and .EXE files) loaded into the application's address space. The RTE also provides distributed, thread-local stack storage for contextual information across interface calls.

Access to configuration information stored in the application binary. The RTE provides a set of functions for accessing information in the configuration record created by the binary rewriter. The RTE, in cooperation with the information logger, provides other Coign components with persistent storage through the configuration record.

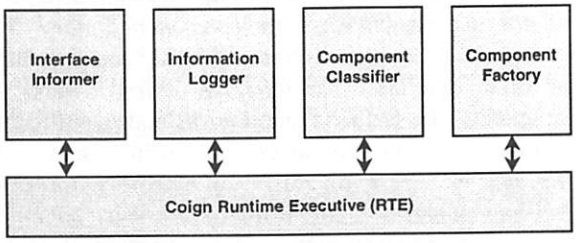


Figure 2. Coign Runtime Architecture.
Runtime components can be replaced to produce lightweight instrumentation, to log component activity, or to remote component instantiation.

3.2. Interface Informer.

The interface informer manages static interface metadata. Other Coign components use data from the interface informer to determine the static type of COM interfaces, and walk input and output parameters of interface function calls. The interface informer also aids the RTE to track the owner component for each interface [20].

The current Coign runtime contains two interface informers. The first interface informer operates during scenario-based profiling. The *profiling informer* uses format strings and interface marshaling code generated by the Microsoft IDL compiler [31] to analyze all function call parameters and precisely measure inter-component communication. Profiling currently adds up to 85% to application execution time (although in most cases the overhead is closer to 45%). Most of this overhead is directly attributable to the interface informer.

The second interface informer remains in the application after profiling to produce the distributed application. The *distribution informer* only examines function call parameters enough to identify interface pointers. Due to aggressive optimization of static interface metadata, the distribution informer imposes an overhead on execution time of less than 3%.

3.3. Information Logger

The information logger summarizes and records data for distributed partitioning analysis. Under direction of the RTE, Coign components pass information about application events to the information logger. Events include component instantiations, component destructions, interface instantiations, interface destructions, and interface calls. The logger is free to process the events as needed. Depending on the logger's implementation, it may ignore the events, write the events to a log file on disk, or accumulate information about the events into in-memory data structures.

The current implementation of the Coign runtime contains three separate information loggers. The *profiling logger,* summarizes data describing inter-component communication into in-memory data structures. At the end of execution, these data structures are written to disk for post-profiling analysis. The profiling logger reduces memory overhead by summarizing data for messages in common size ranges (successive ranges grow in size exponentially). Summarization preserves network independence while significantly lowering storage requirements for communication profiles. The *event logger* creates detailed traces of all component-related events during application execution. A colleague has used logs from the event logger to drive detailed application simulations. During distributed execution, the *null logger* ignores all event log requests.

3.4. Instance classifier

The instance classifier identifies component instances with similar communication profiles across separate executions of an application. Automatic distributed partitioning depends on the accurate prediction of instance communication behavior. Accurate prediction is very difficult for dynamic, commercial application. The classifier groups instances with similar instantiation histories. The classifier operates on the

[1]Our inline redirection and binary-rewriting tools for Windows NT are available separately [21].

theory that two instances created under similar circumstances will exhibit similar behavior (*i.e.* communicate equivalently with the same peers). Part of the output of the profile analysis engine is a map of instance classifications to computers in the network.

Coign currently includes seven instance classifiers although only one, the *internal-function called-by classifier*, is typically used. The best classifiers group instances of the same static type created from the same stack back-trace (call chain). Figure 3 illustrates each classifier.

Program Control Flow:

```
A::V() { ... a->W() ... }
A::W() { ... b1->X() ... }
B::X() { ... b2->Y() ... }
B::Y() { ... c->Z() ... }
C::Z() { ... CoCreateInstance(D) }
```

where:
a is an instance of component class **A**,
b1 and **b2** are instances of component class **B**,
c is an instance of component class **C**,

Classifier Descriptors:

Incremental Classifier:
`[10]` (for 10th call to `CoCreateInstance`)

Procedure Called-By (PCB) Classifier:
`[C::Z, B::Y, B::X, A::W, A::V]`

Static-Type (ST) Classifier:
`[D]`

Static-Type Called-By (STCB) Classifier:
`[D, C, B, B, A]`

Internal-Function Called-By (IFCB) Classifier:
`[D, [c,Z], [b2,Y], [b1,X], [a,W], [a,V]]`

Entry-Point Called-By (EPCB) Classifier:
`[D, [c,Z], [b2,Y], [b1,X], [a,V]]`

Instantiated-By (IB) Classifier:
`[D, c]`

Figure 3. Summary of Classifiers.
Each instance classifier creates a descriptor at instantiation time to uniquely identify groups of similar component instances. Call-chain-based classifiers form a descriptor by examining the execution call stack.

The *incremental classifier* assigns each instance to a different classification based on its order of instantiation during application execution. Serving as a straw man for comparison, the incremental classifier can be expected to perform poorly on commercial, input-driven applications.

The *procedure called-by (PCB) classifier*, similar to Barrett and Zorn's classifier for lifetime prediction in memory allocators [3], groups instances with similar static type and instantiation stack back-trace. When walking the stack, the PCB classifier does not differentiate between individual instances of the same component class.

The *static-type (ST) classifier* groups instances with common component class (static type). The ST classifier cannot differentiate between instances of the same class and must therefore assign all instances to the same machine during distribution. This is a debilitating feature for all of the applications we examined.

The *static-type called-by (STCB) classifier* groups instances by component class and the component classes of instances in the stack back-trace.

The *internal-function called-by (IFCB) classifier* groups instances by their component class and the set of function and instance-classification pairs in the stack back-trace.

The *entry-point called-by classifier* groups instances by their component class and the set of function and instance-classification pairs used to enter each component instance on the stack back-trace.

The depth of the stack back-trace for the PCB, STCB, IFCB, and EPCB classifiers can be tuned to evaluate tradeoffs between accuracy and overhead.

The *instantiated-by classifier* groups instances by their component class and their "parent" (the instance classification from which they were instantiated). The *instantiated-by classifier* is functionally equivalent to the IFCB classifier with a depth-1 stack back-trace.

3.5. Component Factory

The component factory produces a distributed application by manipulating instance placement. Using output from the instance classifier and the profile analysis engine, the component factory moves each component instantiation request to the appropriate computer within the network. During distributed execution, a copy of the component factory is replicated onto each machine. The component factories act as peers. Each traps component instantiation requests on its own machine, forwards requests to other machines as appropriate, and fulfills instantiation requests destined for its machine by invoking COM to create the new component instance. The job of the component factory is straightforward because the instance classifier identifies components for remote placement and DCOM handles message transport. Coign currently contains a symbiotic pair of component factories. Used simultaneously, the first factory handles communication with peer factories on remote machines while the second factory interacts with the instance classifier and the interface informer.

4. Experimental Results

Our experimental environment consists of a pair of 200 MHz Pentium PCs with 32MB of RAM, running

Windows NT 4.0 Service Pack 3. During distributed experiments, the PCs were connected through an isolated 10BaseT Ethernet with Intel EtherExpress Pro cards.

4.1. Application and Scenario Suite

For our experiments, we use a suite of three existing applications built from COM components. The applications employ between a dozen and 150 component classes and range in size from approximate 40,000 to 1.8 million lines of source code. The applications apply a broad spectrum of COM implementation idioms. We believe that these applications represent a wide class of COM applications.

Microsoft PhotoDraw 2000 [32]. PhotoDraw is a consumer application for manipulating digital images. Taking input from high-resolution, color-rich sources such as scanners and digital cameras, PhotoDraw produces output such as publications, greeting cards, or collages. PhotoDraw includes tools for selecting a subset of an image, applying a set of transforms to the subset, and inserting the transformed subset into another image. PhotoDraw was a non-distributed application composed of approximately 112 COM component classes in 1.8 million lines of C++ source code.

Octarine. Octarine is a word-processing application developed by another group at Microsoft Research. Designed as a prototype to explore the limits of component granularity, Octarine contains approximately 150 classes of components. Octarine's components range in granularity from less than 32 bytes to several megabytes. Components in Octarine range in functionality from user-interface buttons to generic object dictionaries to sheet music editors. Octarine manipulates three major types of documents: word-processing, sheet music, and table. Fragments of any of the three document types can be combined into a single document. Octarine is composed of approximately 120,000 lines of C and 500 lines of *x86*-assembly source code.

Corporate Benefits Sample [30]. The Corporate Benefits Sample is an application distributed by the Microsoft Developer Network to demonstrate the use of COM to create 3-tier client-server applications. The Corporate Benefits Sample provides windows for modifying, querying, and creating graphical reports on a database of employees and their corporate human-resource benefits. The entire application contains two separate client front-ends and four alternative middle-tier servers. For our purposes, we use a client front-end consisting of approximately 5,300 lines of Visual Basic code and a middle tier server of approximately 32,000 lines of C++ source code with approximately one dozen component classes. Benefits leverages commercial components (distributed in binary form only) such as the graphing component from Microsoft Office [29].

Each of the applications in our test suite is dynamic and user-driven. The number and type of components instantiated in a given execution is determined by user input during execution. For example, a scenario in which a user inserts a sheet music component into an Octarine document will instantiate different components than a scenario in which the user inserts a table component into the document.

To explore the effectiveness of automatic distribution partitioning on component-based applications, our experimental suite consists of several different scenarios for each application. Scenarios range from simple to complex. The intent of the scenarios is to represent realistic usage while fully exercising the components found in the application. Table 1 describes each scenario.

	Scenario	Description
Octarine	o_newdoc	Create text document.
	o_newmus	Create music document.
	o_newtbl	Create table document.
	o_oldtb0	View 5-page table.
	o_oldtb3	View 150-page table.
	o_oldwp0	View 5-page text document.
	o_oldwp3	View 13-page text document.
	o_oldwp7	View 208-page text document.
	o_oldbth	View 5-page text doc. with tables.
	o_offtb3	o_newdoc then o_oldtb3.
	o_offwp7	o_newdoc then o_oldwp3.
	o_bigone	All of the above in one scenario.
PhotoDraw	p_newdoc	Create new image.
	p_newmsr	Create new composition.
	p_oldcur	View line drawing.
	p_oldmsr	View composition.
	p_offcur	p_newdoc then p_oldcur.
	p_offmsr	p_newdoc then p_oldmsr.
	p_bigone	All of the above in one scenario.
Benefits	b_vueone	View records for an employee.
	b_addone	Add new employee.
	b_delone	Delete employee.
	b_bigone	All of the above in one scenario.

Table 1. Profiling Scenarios.
Profiling scenarios represent major usage scenarios and instantiate most component classes in each application.

4.2. Instance Classification

As described in Section 3.4, the instance classifier must correlate information from profiling with instantiation requests during distributed execution.

Choosing a metric to evaluate the accuracy of an instance classifier is difficult because we must evaluate how well a profile from one instance (or group of instances) correlates to another instance. In the context of

automatic distributed partitioning, a profile and an instance correlate if they have similar resource usage and similar communication behavior (*i.e.* similar peers and peer-communication patterns).

To quantify communication behavior, we introduce the notion of an instance communication vector. An instance communication vector is an ordered tuple of n real numbers (one for each component instance in the application). Each number quantifies the communication time with another component instance (assuming that the other instance is located remotely). The communication vector can be augmented with additional dimensions representing various resources such as memory and CPU cycles. We compare the correlation between two communication vectors with the vector dot product operator. Two vectors with a dot-product correlation of one have equivalent communication behavior (i.e. they communicate equivalently with the same peers). Two vectors with a dot-product correlation of zero share no common communication behavior.

For automatic distributed partitioning, an instance classifier should identify as many unique instance classifications as possible in profiling scenarios in order to preserve distribution granularity. An instance classifier should also be reliable and stable; it should correctly identify instances with similar communication profiles and instantiation contexts.

To evaluate the instance classifiers, we ran classifiers through all of the scenarios except the `bigone` scenarios for each application to create the instance profiles. We then ran classifiers for each application through the `bigone` scenarios. The `bigone` scenarios are a synthesis of the other scenarios for the application. Because all component instances should correlate closely to prior scenarios, no new instance classifications should result from the `bigone` scenario. Table 2 lists the number of classifications identified by each classifier, the number of new classification identified in the `bigone` scenario, the average number of instances per classification, and the average correlation between instance behavior and chosen profile for the Octarine `bigone` scenario. Table 3 lists the same values for IFCB classifier with limited depth stack walks. (The called-by classifiers in Table 2 walk the complete stack.)

Given only the component's static type as context, the ST classifier cannot distinguish instantiations of the same component class used in radically different contexts. The "straw man" classifier, the incremental classifier, fails to correlate instances in the `bigone` scenarios with profiles from the earlier scenarios. It is strictly limited by the order of application execution and user input. Note that incremental classifier would perform well for static applications, but fails miserably for dynamic, commercial applications.

The call-chain-based instance classifiers (PCB, STCB, IFCB, EPCB, and IB) preserve more distribution granularity because they take into account contextual information when classifying an instantiation. The STCB, IFCB and EPCB classifiers are similar in accuracy. They differ however, in the number of unique component classifications they identify. As would be expected, the IFCB classifier, which uses the largest amount of contextual information, identifies the largest number of classifications.

Instance Classifier	Profiled Classifications	New (`bigone`) Classifications	Ave. Instances / Classification	Average Correlation
Incremental	1090	2561	1.0	0.225
Procedure Called-By	1262	0	2.9	0.766
Static-Type	80	0	45.6	0.574
Static-Type Called-By	713	0	5.1	0.809
Internal-Func. Called-By	1434	0	2.6	0.848
Entry-Pointer Called-By	1032	0	3.5	0.829
Instantiated-By	590	0	6.2	0.809

Table 2. Classifier Accuracy.
Classifiers with a higher number of classifications recognize more unique component instances. Those with a higher average correlation are more accurate.

Internal-Function Called-By Classifier Stack-Walk Depth	Profiled Classifications	Ave. Instances / Classification	Average Correlation
1	590	6.2	0.809
2	977	3.7	0.829
3	1184	3.1	0.848
4	1383	2.6	0.848
8	1434	2.6	0.848
16	1434	2.6	0.848
Complete	1434	2.6	0.848

Table 3. Accuracy as a Function of Stack Depth.
Both classifier accuracy (average correlation) and number of classifications increase with the depth of the stack walked.

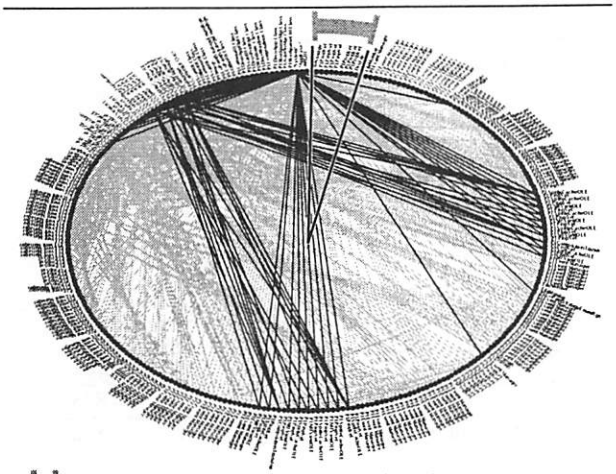
Fundamentally, our instance classifiers are limited in their accuracy by the amount of contextual information available before a component is instantiated. They cannot differentiate two instances with identical instantiation context, but vastly different communication profiles. However, experimental evidence suggests the STCB, IFCB, EPCB, and IB classifiers preserve distribution granularity and correlate profiles with sufficient

accuracy to enable automatic distributed partitioning of commercial applications.

4.3. Distributions

Because Coign makes distribution decisions at component boundaries, it success depends on programmers to build applications with significant numbers of components. To evaluate Coign's effectiveness in automatically creating distributed applications, we ran each application in the test suite through a simple profiling scenario consisting of the simplest practical usage of the application. After profiling, Coign partitioned each application between a client and server of equal compute power on an isolated 10BaseT Ethernet network. For simplicity, we assume there is no contention for the server.

Figure 4 plots the distribution of PhotoDraw. In the profiling scenario, PhotoDraw loads a 3MB graphical composition from storage, displays the image, and exits. Of 295 components in the application, eight are placed on the server. One of the components placed on the server reads the document file. The other seven components are high-level *property sets* created directly from data in the file; with larger input sets than output sets, they are placed on the server to reduce communication.

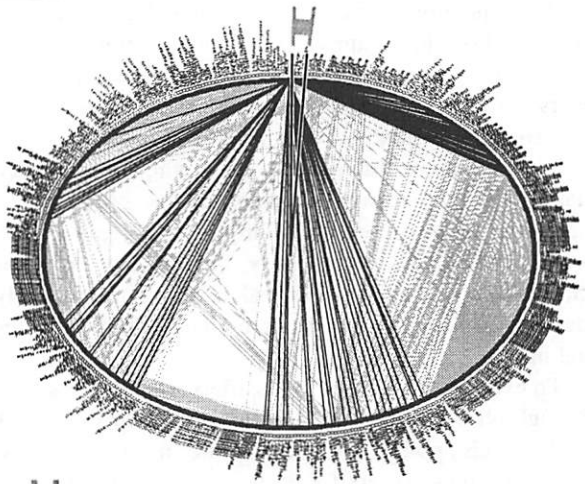


= *Component instances placed on Server.*

Figure 4. PhotoDraw Distribution.
Of 295 components in the application, Coign places eight on the server. Black lines represent non-distributable interfaces between components. Gray lines represent distributable interfaces.

As can be seen in Figure 4, PhotoDraw contains many significant interfaces (almost 50) that can not be distributed (shown as solid black lines). The most important non-distributable interfaces connect the *sprite cache* components (on the bottom and right) with user interface components (on the top left). Each sprite cache manages the pixels for a hierarchical subset of an image in the composition. Most of the data passed between sprite caches moves through shared memory regions. Pointers to the shared-memory regions are passed opaquely through non-distributable interfaces.

While Coign can extract a functional distribution from PhotoDraw, most of the distribution granularity in the application is hidden by non-distributable interfaces. To enable other, potentially better distributions, either the non-distributable interfaces in PhotoDraw must be replaced with distributable IDL interfaces, or Coign must be extended to support transparent migration of shared memory regions; in essence leveraging the features of software distributed-shared memory [26].



= *Component instances placed on Server.*

Figure 5. Octarine Distribution.
Of 458 components in the application, Coign places two on the server. Most of the non-distributable interfaces in Octarine connect elements of the GUI.

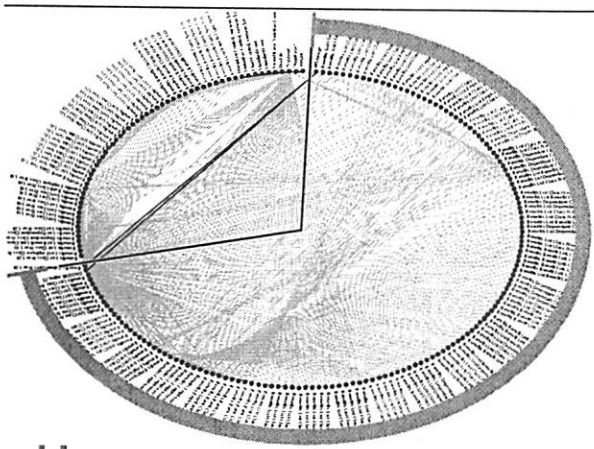
Figure 5 shows the distribution of the Octarine word processor. In this scenario, Octarine loads and displays the first page of a 35-page, text-only document. Coign places only two components of 458 on the server. One of the components reads the document from storage; the other provides information about the properties of the text to the rest of the application. While Figure 5 contains many non-distributable interfaces, these interfaces connect components of the GUI, and are not directly related to the document file. Unlike the other applications in our test suite, Octarine's GUI is composed of literally hundreds of components. It is highly unlikely that these GUI components would ever be located on the server. Direct document-related processing for this scenario is limited to just 24 components.

Figure 6 contains the distribution for the MSDN Corporate Benefits Sample. As shipped, Benefits can be distributed as either a 2-tier or a 3-tier client-server

application. The 2-tier implementation places the Visual Basic front-end and the business-logic components on the client and the database, accessed through ODBC [28], on the server. The 3-tier implementation places the front-end on the client, the business-logic on the middle tier, and the database on the server. Coign cannot analyze proprietary connections between the ODBC driver and the database server. We therefore focus our analysis on the distribution of components in the front end and middle tier of the 3-tier implementation.

Coign analysis shows that application performance can be improved by moving some of the middle-tier components into the client. The distribution chosen by Coign is quite surprising. Of 196 components in the client and middle tier, Coign places 135 on the middle tier versus 187 chosen by the programmer. The new distribution reduces communication by 35%.

The intuition behind the new distribution is that many of the middle-tier components cache results for the client. Coign moves the caching components, but not the business-logic itself, from the middle-tier to the client. Although not used in this analysis, the programmer can place two kinds of explicit location constraints on components to guarantee data integrity and security requirements. Absolute constraints explicitly force an instance to a designated machine. Pair-wise constraints force the co-location of two component instances.



= *Component instances placed on Server.*

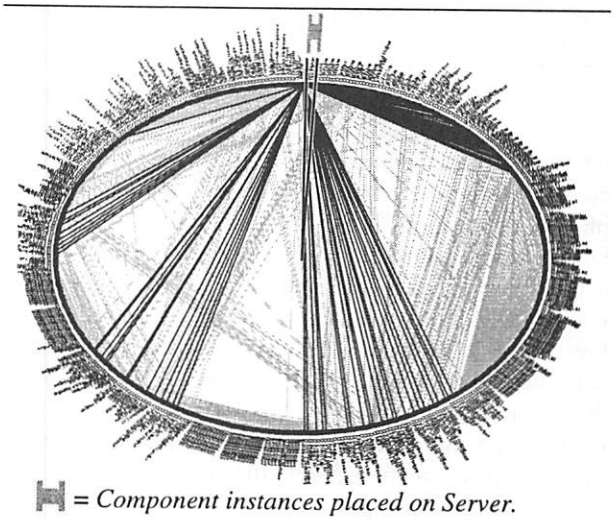
Figure 6. Corporate Benefits Distribution.
Of 196 components in the client and middle tier, Coign places 135 of the components on the middle tier where the programmer placed 187.

The programmer's distribution is a result of two design decisions. First, the middle tier represents a conceptually clean separation of business logic from the other pieces of the application. Second, the front-end is written in Visual Basic, an extremely popular language for rapid development of GUI applications, while the business logic is written in C++. It would be awkward for the programmer to create the distribution easily created by Coign.

The Corporate Benefits Sample demonstrates that Coign can improve the distribution of applications designed by experienced client-server programmers. In addition to direct program decomposition, Coign can also selectively enable per-interface caching (as appropriate) through COM's semi-custom marshaling mechanism.

4.4. Changing Scenarios and Distributions

The simple scenarios in the previous section demonstrate that Coign can automatically choose a partition and distribute an application. The Benefits example notwithstanding, one could argue that an experienced programmer with appropriate tools could partition the application at least as well manually. Unfortunately, a programmer's best-effort manual distribution is static; it cannot readily adapt to changes in network performance or user-driven usage patterns. In the realm of changing environments, Coign has a distinct advantage as it can repartition and distribute the application arbitrarily often. In the limit, Coign can create a new distributed version of the application for each execution.



= *Component instances placed on Server.*

Figure 7. Octarine with Multi-page Table.
With a document containing a five-page table, Coign locates only a single component on the server.

The merits of a distribution customized to a particular usage pattern are not merely theoretical. Figure 7 plots the optimized distribution for Octarine loading a document containing a single, 5-page table. For this scenario, Coign places only a single component out of 476 on the server. The results are comparable to those of Octarine loading a document containing strictly text (Figure 5). However, if fewer than a dozen small tables

are added to the 5-page text document, the optimal distribution changes radically. As can be seen in Figure 8, Coign places 281 out of 786 components on the server. The difference in distribution is due to the complex negotiations for page placement between the table components and the text components. Output from the page-placement negotiation to the rest of the application is minimal.

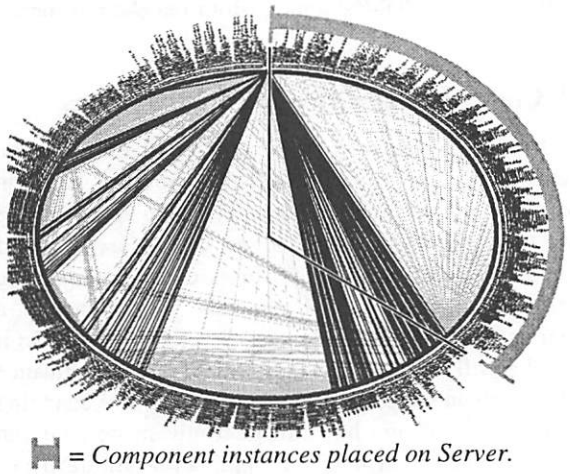


Figure 8. Octarine with Tables and Text.
With a five-page document containing fewer than a dozen embedded tables, Coign places 281 of 786 application components on the server.

In a traditional distributed system, the programmer would likely optimize the application for the most common usage pattern. At best, the programmer could embed a minimal number of distribution alternatives into the application. With Coign, the programmer need not favor one distribution over another. The application can be distributed with an inter-component communication model optimized for the most common scenarios. Over the installed lifetime of the application, Coign can periodically re-profile the application and adjust the distribution accordingly. Even without updating the inter-component communication model, Coign can adjust to changes in application infrastructure, such as the relative computation power of the client and server, or network latency and bandwidth.

4.5. Performance of Chosen Distributions

Table 4 lists the communication time for each of the application scenarios. The default distribution is the distribution of the application as configured by the developer without Coign. For both the default and Coign-chosen distributions, data files are placed on the server. As can be seen, Coign never chooses a worse distribution than the default. In the best case, Coign reduces communication time by 99%. The Corporate Benefits Application has significant room for improvement as suggested by the change in its distribution in Section 4.3.

The results suggest that Coign is better at optimizing existing distributed applications than creating new distributed applications from desktop applications. The distribution of desktop COM-based applications is limited by the extensive use of non-remotable interfaces. For PhotoDraw in particular, Coign is severely constrained by the large number of non-remotable interfaces. It is important to note that the distributions available in Octarine and PhotoDraw are not limited by the granularity of their components, but by their interfaces. We believe that as the development of component-based applications matures, developers will learn to create interfaces with better distribution properties, thus strengthening the benefits of Coign.

Scenario	Comm. Time (secs.) Default	Coign	Savings
o_newdoc	0.152	0.152	0%
o_newmus	0.149	0.149	0%
o_newtbl	0.006	0.006	0%
o_oldtb0	1.058	1.048	1%
o_oldtb3	15.064	0.042	99%
o_oldwp0	0.143	0.143	0%
o_oldwp3	0.696	0.696	0%
o_oldwp7	21.089	1.099	95%
o_oldbth	1.734	0.562	68%
o_offtb3	15.079	0.037	99%
o_offwp7	20.878	1.090	95%
o_bigone	27.497	22.630	18%
p_newdoc	4.726	4.496	5%
p_newmsr	17.016	15.014	12%
p_oldcur	2.384	1.613	32%
p_oldmsr	14.517	11.482	21%
p_offcur	1.583	0.722	54%
p_offmsr	14.650	11.497	22%
p_bigone	33.032	27.084	18%
b_vueone	1.465	0.954	35%
b_addone	2.322	1.601	31%
b_delone	3.414	2.834	17%
b_bigone	1.754	1.414	19%

Table 4. Reduction in Communication Time.
Communication time for the default distribution of the application (as shipped by the developer) and for the Coign-chosen distribution.

4.6. Accuracy of Prediction Models

To verify the accuracy of Coign's model of application communication time and execution time, we compare the predicted execution time for each scenario with the measured execution time (Table 5). In each case, the application is optimized for the chosen scenario before execution. Many of the scenarios had no signifi-

cant difference between predicted and actual execution time; only seven had an error of 5% or greater, and none varied by more than 8%. From these measurements, we conclude that Coign's model of application communication and execution time is sufficiently accurate to warrant confidence in the distributions chosen by Coign's graph-cutting algorithm.

Scenario	Execution Time (sec.) Predicted	Measured	Error
o_newdoc	10.7	10.7	0%
o_newmus	10.9	10.9	0%
o_newtbl	9.3	9.3	0%
o_oldtb0	19.0	19.1	0%
o_oldtb3	231.1	231.1	0%
o_oldwp0	5.5	5.7	-3%
o_oldwp3	7.2	7.3	-2%
o_oldwp7	33.4	33.6	-1%
o_oldbth	33.6	33.6	0%
o_offtb3	232.7	232.7	0%
o_offwp7	67.2	65.6	2%
o_bigone	416.1	429.7	-3%
p_newdoc	14.3	14.3	0%
p_newmsr	76.8	72.9	5%
p_oldcur	18.8	18.8	0%
p_oldmsr	49.0	49.5	-1%
p_offcur	18.1	18.1	0%
p_offmsr	53.8	54.2	-1%
p_bigone	139.6	136.3	2%
b_vueone	9.4	8.9	6%
b_addone	14.6	13.9	5%
b_delone	8.9	8.4	7%
b_bigone	5.6	5.2	8%

Table 5. Accuracy of Prediction Models.
Predicted application execution time and measured application execution time for Coign distributions.

5. Related Work

The idea of automatically partitioning and distributing applications is not new. The Interconnected Processor System (ICOPS) [27, 40, 41] supported distributed application partitioning in the 1970's. ICOPS pioneered the use of compiler-generated stubs for interprocess communication. ICOPS was the first system to use scenario-based profiling to gather statistics for distributed partitioning; the first system to support multiple distributions per application based on host-processor load; and the first system to use a minimum-cut algorithm [11] to choose distributions. ICOPS distributed HUGS, a co-developed, two-dimensional drafting program. HUGS consisted of seven modules. Three of these—consisting of 20 procedures in all—could be located on either the client or the server.

Unlike Coign, which can distributed individual component instances, ICOPS was procedure-oriented. ICOPS placed all instances of a specific class on the same machine; a serious deficiency for commercial applications. Tied to a single language and compiler, ICOPS relied on metadata generated by the compiler to facilitate transfer of data and control between computers. Modules compiled in another language (or by another compiler) could not be distributed because they did not contain appropriate metadata. ICOPS gave the application the luxury of location transparency, but still required the programmer or user to explicitly select a distribution based on machine load.

Configurable Applications for Graphics Employing Satellites (CAGES) [16, 17] allowed a programmer to develop an application for a single computer and later distribute the application across a client/server system. Unlike ICOPS, CAGES did not support automatic distributed partitioning. Instead, the programmer provided a pre-processor with directions about where to place each program module. The programmer could change a distribution only after recompiling the application with a new placement description file. Like ICOPS, CAGES was procedure-oriented; programs could be distributed at the granularity of procedural modules in the PL/I language. The largest application distributed by CAGES consisted of 28 modules. To aid the programmer in choosing a distribution, CAGES produced a "nearness" matrix through static analysis. The "nearness" matrix quantified the communication between modules, thus hinting how "near" the modules should be placed to each other.

One important advantage of CAGES over ICOPS was its support for simultaneous computation on both the satellite and the host computers. CAGES provided the programmer with the abstraction of one dual-processor computer on top of two physically disjoint single-processor computers. The CAGES runtime provided support for RPC and asynchronous signals.

Both ICOPS and CAGES were severely constrained by their granularity of distribution: the PL/I or ALGOL-W procedural module. Neither system ever distributed an application with more than a few dozen modules. However, despite their weaknesses, each system provided some degree of support for automatic or semi-automatic distributed application partitioning.

The Intelligent Dynamic Application Partitioning (IDAP) system [22, 25], an ADPS for CORBA applications, is an add-on to IBM's VisualAge Generator. Using VisualAge Generator's visual builder, a programmer designs an application by instantiating and connecting components in a graphical environment. The builder emits code for the created application.

The "dynamic" IDAP name refers to the usage of scenario-based profiling as an alternative to static analysis. IDAP first generates a version of the application with an instrumented message-passing system. IDAP runs the instrumented application under control of a test facility with the VisualAge system. After ap-

plication execution, the programmer either manually partitions the components or invokes an automatic graph-partitioning algorithm. The algorithm used is an approximation algorithm capable of multi-way cuts for two or more hosts [10]. After choosing a distribution, VisualAge generates a new version of the application. The IDAP developers have tested their system on several real applications, but in each case, the application had "far fewer than 100" components [25].

IDAP supports distributed partitioning only for statically instantiated components. IDAP requires full access to source code. Another potential restriction is the natural granularity of CORBA applications. CORBA components tend to be large-grained objects whereas COM components in the applications we examined have a much smaller granularity. Often each CORBA component must reside in a separate server process. In essence, IDAP helps the programmer decide where CORBA servers should be placed in a network, but does not facilitate program decomposition. The IDAP programmer must be very aware of distribution choices. IDAP helps the user to optimize the distribution, but does not raise the level of abstraction above the distribution mechanisms. With a full-featured ADPS, such as Coign, the programmer can focus on component development and leave distribution to the system.

5.1. Distributed Object Systems

Emerald [5, 6] combines a language and operating system to create an object-oriented system with first class support for distribution. Emerald objects can migrate between machines during execution; they can also be fixed to a particular machine, or be co-located under programmer control through language operators [23]. Emerald is limited to a single language and does not attempt to automatically place objects to minimize application communication.

The SOS [39], Globe [19], and Legion [14] distributed object systems provide true location-transparent objects and direct programmer control over object location. Globe and Legion each anticipate scaling to the entire Internet. However, none of these systems supports automatic program modification to minimize communication.

5.2. Parallel Partitioning and Scheduling

Strictly speaking, the problem of distributed partitioning is a proper subset of the general problem of parallel partitioning and scheduling. Our work differs from similar work in parallel scheduling ([24, 36-38]) in two primary respects. First, Coign accommodates applications in which components are instantiated and destroyed dynamically throughout program execution. Traditional parallel partitioning focuses on static applications. Second, because Coign operates on binary applications, it can optimize application without access to source code (a necessary feature in the domain of commercial component-based applications).

Coign does not increase the parallelism in application code, nor does it perform horizontal load-balancing between peer servers. Instead, Coign focuses on "vertical" load-balancing within the application. The question of how to minimize communication and maximize parallelism in large dynamic, commercial applications remains open.

6. Conclusions and Future Work

Coign is the first ADPS to distribute binary applications and the first ADPS to partition applications with dynamically instantiated components of any kind (either binary or source). Dynamic component instantiation is an integral feature of modern desktop applications. One of the major contributions of our work is a set of dynamic instance classifiers that correlate newly instantiated components to similar instances identified during scenario-based profiling.

Evaluation of Coign shows that it minimizes distributed communication time for each of the applications and scenarios in our test suite. Surprisingly, the greatest reduction in communication time occurs in the distributed Corporate Benefits Sample where Coign places almost half of the middle-tier components on the client without violating application security. Results from Octarine demonstrate the potential for more than one distribution of an application depending on the user's predominant document type.

We envision two models for Coign to create distributed applications. In the first model, Coign is used with other profiling tools as part of the development process. Coign shows the developer how to distribute the application optimally and provides the developer with feedback about which interfaces are communication "hot spots." The programmer fine-tunes the distribution by enabling custom marshaling and caching on communication intensive interfaces. The programmer can also enable or disable specific distributions by inserting or removing location constraints on specific components and interfaces. Alternatively, the programmer can create a distributed application with minimal effort simply by running the application through profiling scenarios and writing the corresponding distribution model into the application binary without modifying application sources.

In the second usage model, Coign is applied onsite by the application user or system administrator. The user enables application profiling through a simple GUI to Coign. After "training" the application to the user's usage patterns—by running the application through

representative tasks with profiling—the GUI triggers post-profiling analysis and writes the distribution model into the application. In essence, the user has created a customized version of the distributed application without any knowledge of the underlying details.

In the future, Coign could automatically decide when usage differs significantly from profiled scenarios and silently enable profiling to re-optimize the distribution. The Coign runtime already contains sufficient infrastructure to allow "fully automatic" distribution optimization. The lightweight version of the runtime, which relocates component instantiation requests to produce the chosen distribution, could count messages between components with only slight additional overhead. Run time message counts could be compared with related message counts from the profiling scenarios to recognize changes in application usage.

References

[1] Aral, Ziya, Illya Gertner, and Greg Schaffer. Efficient Debugging Primitives for Multiprocessors. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 87-95. Boston, MA, April 1989.

[2] Arnold, Thomas R., II,. *Software Testing with Visual Test 4.0*. IDG Books Worldwide, Foster City, CA, 1996.

[3] Barrett, David A. and Benjamin G. Zorn. Using Lifetime Predictors to Improve Memory Allocation Performance. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 187-196. Albuquerque, NM, June 1993. ACM.

[4] Birrell, A. D. and B. J. Nelson. Implementing Remote Procedure Call. *ACM Transactions on Computer Systems*, 2(1):39-59, 1984.

[5] Black, A., N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 78-86. Portland, OR, October 1986.

[6] Black, A., N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65-76, 1987.

[7] Bokhari, Shahid. Partitioning Problems in Parallel, Pipelined, and Distributed Computing. *IEEE Transactions on Computers*, 37(1):48-57, 1988.

[8] Brown, Nat and Charlie Kindel. *Distributed Component Object Model Protocol -- DCOM/1.0*. Microsoft Corporation, Redmond, WA, 1996.

[9] Cormen, Thomas H., Charles E. Leiserson, and Rondald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[10] Dahlhaus, E., D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The Complexity of Multiterminal Cuts. *SIAM Journal on Computing*, 23(4):864-894, 1994.

[11] Ford, Lester R., Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

[12] Gary, Naveen, Vijay V. Vazirani, and Mihalis Yannakakis. Multiway Cuts in Directed and Node Weighted Graphs. *Proceedings of the 21st International Colloquim on Automata, Languages, and Programming (ICALP)*, pp. 487-498. Jerusalem, Isreal, July 1994. Springer-Verlag.

[13] Goldberg, Andrew V., Éva Tardos, and Robert E. Tarjan. Network Flow Algorithms. Computer Science Department, Stanford University, Technical Report STAN-CS-89-1252, 1989.

[14] Grimshaw, Andrew S., William A. Wulf, and the Legion Team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), 1997.

[15] Hamilton, K. G. A Remote Procedure Call System. Ph. D. Dissertation, Computer Laboratory TR 70. University of Cambridge, Cambridge, UK, 1984.

[16] Hamlin, Griffith, Jr. Configurable Applications for Satellite Graphics. *Proceedings of the Third Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '76)*, pp. 196-203. Philadelphia, PA, July 1976. ACM.

[17] Hamlin, Griffith, Jr. and James D. Foley. Configurable Applications for Graphics Employing Satellites (CAGES). *Proceedings of the Second Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '75)*, pp. 9-19. Bowling Green, Ohio, June 1975. ACM.

[18] Hartman, D. Unclogging Distributed Computing. *IEEE Spectrum*, 29(5):36-39, 1992.

[19] Homburg, Philip, Martin van Steen, and Andrew S. Tanenbaum. An Architecture for a Scalable Wide Area Distributed System. *Proceedings of the Seventh ACM SIGOPS European Workshop*. Connemara, Ireland, September 1996.

[20] Hunt, Galen. Automatic Distributed Partitioning of Component-Based Applications. Ph.D. Dissertation, Department of Computer Science. University of Rochester, 1998.

[21] Hunt, Galen. Detours: Binary Interception of Win32 Functions. Microsoft Research, Redmond, WA, MSR-TR-98-33, July 1998.

[22] IBM Corporation. *VisualAge Generator*. Version 3.0, Raleigh. NC, 1997.

[23] Jul, Eric, Henry Levy, Norman Hutchinson, and Andew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109-133, 1988.

[24] Kennedy, Ken and Ajay Sethi. A Communication Placement Framework for Unified Dependency and

Data-Flow Analysis. *Proceedings of the Third International Conference on High Performance Computing*. India, December 1996.

[25] Kimelman, Doug, Tova Roth, Hayden Lindsey, and Sandy Thomas. A Tool for Partitioning Distributed Object Applications Based on Communication Dynamics and Visual Feedback. *Proceedings of the Advanced Technology Workshop, Third USENIX Conference on Object-Oriented Technologies and Systems*. Portland, OR, June 1997.

[26] Li, Kai and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *Transactions on Computer Systems*, 7(4):321-359, 1989.

[27] Michel, Janet and Andries van Dam. Experience with Distributed Processing on a Host/Satellite Graphics System. *Proceedings of the Third Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '76)*, pp. 190-195. Philadelphia, PA, July 1976.

[28] Microsoft Corporation. *Microsoft Open Database Connectivity Software Development Kit.* Version 2.0. Microsoft Press, 1994.

[29] Microsoft Corporation. *Microsoft Office 97.* Version 6.0, Redmond, WA, 1997.

[30] Microsoft Corporation. Overview of the Corporate Benefits System. *Microsoft Developer Network*, 1997.

[31] Microsoft Corporation. *MIDL Programmer's Guide and Reference*. Windows Platform SDK, Redmond, WA, 1998.

[32] Microsoft Corporation. *PhotoDraw 2000.* Version 1.0, Redmond, WA, 1998.

[33] Naor, Joseph and Leonid Zosin. A 2-Approximation Algorithm for the Directed Multiway Cut Problem. *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science*, pp. 548-553, 1997.

[34] Nelson, B. J. Remote Procedure Call. Ph.D. Dissertation, Department of Computer Science. Carnegie-Mellon University, 1981.

[35] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.0.* vol. Revision 2.0, Framingham, MA, 1995.

[36] Ousterhout, J. K. Techniques for Concurrent Systems. *Proceedings of the Third International Conference on Distributed Computing Systems*, pp. 22-30. Miami/Ft. Lauderdale, FL, October 1982. IEEE.

[37] Polychronopolous, C. D. *Parallel Programming and Compilers*. Kluwer Academic Publishers, Boston, MA, 1988.

[38] Sarkar, Vivek. Partitioning and Scheduling for Execution on Multiprocessors. Ph.D. Dissertation, Department of Computer Science. Stanford University, 1987.

[39] Shapiro, Marc. Prototyping a Distributed Object-Oriented Operating System on Unix. *Proceedings of the Workshop on Experiences with Distributed and Multiprocessor Systems*, pp. 311-331. Fort Lauderdale, FL, October 1989. USENIX.

[40] Stabler, George M. A System for Interconnected Processing. Ph.D. Dissertation, Department of Applied Mathematics. Brown University, Providence, RI, 1974.

[41] van Dam, Andries, George M. Stabler, and Richard J. Harrington. Intelligent Satellites for Interactive Graphics. *Proceedings of the IEEE*, 62(4):483-492, 1974.

[42] Vinoski, Steve. CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments. *IEEE Communications*, 14(2), 1997.

Tapeworm: High-Level Abstractions of Shared Accesses

Peter J. Keleher
keleher@cs.umd.edu
Department of Computer Science
University of Maryland
College Park, MD 20742

We describe the design and use of the tape *mechanism, a new high-level abstraction of accesses to shared data for software DSMs. Tapes can be used to "record" shared accesses. These recordings can be used to predict future accesses. Tapes can be used to tailor data movement to application semantics. These data movement policies are layered on top of existing shared memory protocols.*

We have used tapes to create the Tapeworm prefetching library. Tapeworm implements sophisticated record/replay mechanisms across barriers, augments locks with data movement semantics, and allows the use of producer-consumer segments, which move entire modified segments when any portion of the segment is accessed. We show that Tapeworm eliminates 85% of remote misses, reduces message traffic by 63%, and improves performance by an average of 29% for our application suite.

1. Introduction

This paper introduces the notion of *tapes*: a new high-level abstraction that allows applications to achieve better performance on distributed shared memory (DSM) protocols. DSM protocols support the abstraction of shared memory to parallel applications running on networks of workstations. The DSM abstraction provides an intuitive programming model and allows applications to become portable across a broad range of environments. However, this level of abstraction prevents the application from improving performance by explicitly directing data movement. While it is relatively easy to get parallel applications working on current DSMs, it can be very difficult to achieve high performance.

Tapes make this task easier by allowing the data movement to be directed by the application at a high level of abstraction. A tape is essentially an object that encapsulates an arbitrary number of updates to shared data. Tapes are created through recording of updates to shared data made by the local process. Once created, a tape provides a convenient way to manipulate the updates. The data referenced by a tape can be sent to another process. Tapes can be reshaped by changing the set of data to which they refer. Tapes can also be added and subtracted, allowing a single tape to describe any arbitrary set of updates.

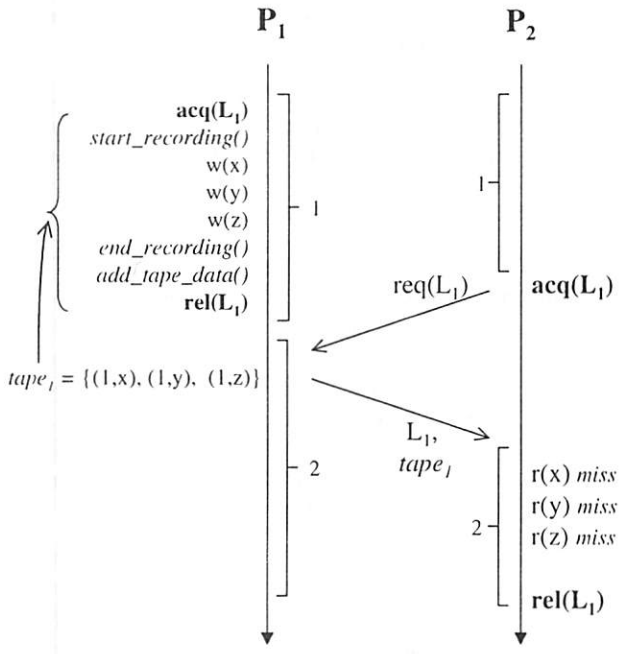


Figure 1: Tapes: $tape_1$ describes the writes performed by P_1. Subsequent misses by P_2 can be avoided if the tape, together with the data it describes, is transferred with the lock.

As a quick example, Figure 1 shows a simple use of the tape mechanism. We defer detailed description of this example until the next section. Essentially, however, the example shows process P_1 modifying three shared pages while holding lock L_1, followed by P_2 acquiring the same lock and reading the same three pages.

In a traditional invalidate protocol, P_1's modifications would cause all three pages to be invalidated at P_2. The subsequent reads by P_2 would each cause remote page faults. Each fault is satisfied by retrieving a current copy of the faulting page from a remote processor, and hence implies at least one network RPC. After the data is returned and copied to the correct location, page protections are changed to allow the page to be accessed normally.

By including the code in italics, however, P_1 can record the accesses automatically, append the modified data to the lock grant message, and *update*, rather than invali-

date, P_2's copy of the page. For each page fault thereby avoided, the system eliminates both local fault-handling overhead and network RPC's.

The key points of this example are the following. First, tapes allow sharing behavior to be captured at runtime. The system needs neither compiler cooperation nor extensive user interaction in order to determine exactly which pieces of shared data are accessed by P_1. This is important because we do not assume any explicit associations between synchronization and shared data, just as no such associations are assumed in a typical multithreaded environment such as PTHREADS.

Second, moving the data with the lock is only a performance optimization, it can not cause correctness to be violated. No damage is done if P_2 does not access either x, y, or z. Any additional pages accessed by P_2 will be demand-paged across the network when the pages are accessed.

While tapes could be used directly by applications, they are probably more useful when folded into specialized synchronization libraries. Such libraries can reduce the total application involvement to just the replacement of calls to generic synchronization primitives with calls to the corresponding routines in the new libraries. This indirection allows the synchronization implementation to be quite simple, without losing any generality.

The primary claimed advantage of DSM systems over message-passing programming models is ease of use. By abstracting away any need to specify data locations, DSM systems allow parallel and distributed applications to be more simply created. Requiring applications to contain additional annotations would seem to run counter to this goal. However, synchronization libraries can hide the mechanism from programmer view. The only change needed to use tape mechanisms in these cases is linking with a different library. Moreover, tape mechanisms can be added to applications incrementally. Applications can be developed and tested without tapes. Since tape mechanisms do not affect correctness, adding tape calls can not break any application that has already been debugged.

We used tapes to implement Tapeworm, a new synchronization library that is layered on top of existing consistency and synchronization protocols in CVM [1], a software distributed shared memory system. The use of tapes allowed us to write Tapeworm in fewer than 400 lines of C++ code. At the same time, Tapeworm is able to track and use very sophisticated data movement patterns. Specifically, Tapeworm augments ordinary locks to include data movement semantics as well as synchronization. Tapeworm also supports producer-consumer regions and record/replay barriers. Record/replay barriers use recordings of data accesses from one iteration of an application to anticipate accesses during future iterations.

Overall, Tapeworm eliminates an average of 85% of data misses on our suite of applications. The reduction in misses translates into a reduction in message traffic of 63%, and an average improvement in overall performance of approximately 29%.

The rest of the paper is as follows. Section 2 discusses the high-level semantics of tapes in a protocol-independent fashion. Section 3 describes Tapeworm, a high-performance synchronization library built using tapes. Section 4 describes the requirements Tapeworm makes on the underlying consistency protocols. Section 5 describes Tapeworm's performance, Section 6 describes related work, and Section 7 concludes.

2. Tape semantics

Tapes are implemented as a software layer that logically resides on top of existing consistency and synchronization protocols. Conceptually, at least, the tape mechanism is independent of both the underlying protocol implementation, and of the precise application access orderings that are being captured. In practice, tapes are particularly well suited for the relaxed consistency models discussed below.

The primary way in which we expect tapes to be used is in augmenting invalidate protocols in software DSMs. Such systems consist of a single thread or process per machine, a shared segment that can be transparently accessed by any of the processes, and at least a rudimentary set of synchronization mechanisms. Synchronization is usually implemented in addition to, rather than on top of, the consistency mechanism. The best page-based consistency mechanisms are based on some form of release consistency (RC) [2] or lazy release consistency (LRC) [3, 4]. Both allow consistency actions to be delayed until subsequent synchronizations. These protocols are ideal for the use of tapes in that they allow considerable freedom as to when data actually moves. Tapes could be used with more strict protocols, but more work would be required to provide the necessary hooks.

Implementation

CVM's implementation of tapes is made efficient by grouping logically related shared accesses into a single access. Each process's execution is divided into distinct intervals, each of which is labeled with a system-unique interval id. The exact method by which intervals are defined is not important, although most protocols will probably delimit intervals by synchronization events. For example, each of the processes in Figure 1 has two

intervals, delimited by synchronization accesses to lock L_1.

Second, all modifications made to a single page during an interval are combined into a single modification. We can then express tapes in terms of lists of modified pages and intervals, instead of addresses and cycle counts[1].

More specifically, a tape consists of a set of *events*, each of which is a tuple (x,y), where x is an interval id and y is a set of page id's. Hence, $tape_1$ in Figure 1 consists of the three events $\{(1,1), (1,2), (1,3)\}$. Note that the event (and the tape) consists only of the tuple, it does not contain the actual modifications. The actual modifications are tracked by the underlying protocol.

The approach shown in this example has several advantages over other approaches described in the literature. Simple update protocols push modified data to existing replicas to *update* them, rather than *invalidating* them. The advantage of such protocols is that subsequent page faults are avoided, but the lack of any selectivity usually causes update protocols to move far more data than invalidate protocols. Several researchers have described more selective update protocol variants [5-7] that might also suffice in this example. However, these protocols effectively encode expected sharing behavior into the underlying protocol. By making such expectations part of the programmable protocol interface, the tape mechanism has far more flexibility.

Clearly, this is not the whole picture. First, P_2 might be missing other modifications to the three pages. However, this could only occur if modifications to other portions of the page (false sharing) are proceeding concurrently. Otherwise, P_1 would have retrieved copies before accessing the pages itself. Second, application programmers are unlikely to want to reason with tapes. However, all tape references could be encapsulated into modified synchronization routines. The sole application-visible change would be in calling a different version of the lock routines.

This example does not show the full generality of the tape mechanism. Since the tape consists internally of events, the accesses need not be to contiguous pages, have occurred at similar times, or have been performed by the same process. Since tapes consist only of abstract descriptions of shared modifications rather than the modifications themselves, they are relatively lightweight and can be stored, transmitted, and otherwise manipulated.

[1] The use of the term "page" throughout this paper is a convention. The units can be of any shape that can be tracked by the underlying consistency protocol.

Tape Creation

Tapes can be created in several different ways, but the primary method is that shown in Figure 1, e.g. recording accesses over a period of time. This method of creating tapes enables synchronization protocols to capture dynamic access patterns at runtime, rather than relying on the programmer or compiler to derive complete information statically.

A second method of creating tapes is for them to be generated by hooks into the underlying consistency protocol. While we defer full discussion of the interface to the underlying protocol until Section 4, `missing_data_tape(Extent *)` is fundamental to some of the interfaces discussed in the next section. Its function is to create and return a tape that describes all updates needed to validate the region of memory described by an *extent*. A shared page is *validated* by applying all updates necessary to bring the page up to date.

Extent is short for "data extent." An extent is an object that names a set of pages. For example, a tape can be flattened into an extent that contains the set of pages accessed by the tape's events. Assume that an extent 'ext' names a large data structure that resides on a set of invalid shared pages. Pages are usually invalid because they have been modified by remote processes, and the remote modifications have yet to be applied locally. A local `missing_data_tape(ext)` call returns a tape that lists every such remote update that is needed to re-validate the invalid pages. This tape can be used to request all of the updates at once, possibly before the data is actually needed. The result is greater latency tolerance, and the potential for greater overlap of communication and computation.

Once a tape has been created, it can be transmitted to remote sites, flattened into an extent, pruned to contain only notices that pertain to a given extent, or added to another tape.

While our discussion of tapes has concentrated on write accesses so far, analogous abstractions can be defined for read accesses, and data requests from other processes.

3. The *Tapeworm* Library

The following subsections describe three types of Tapeworm-based synchronization interfaces that we found useful for our application suite: *record-replay barriers*, *update locks*, and *producer-consumer regions*.

These mechanisms were carefully constructed in order to accommodate weak memory consistencies. We assume lazy release consistency (LRC) in this work, but the same interface will handle almost any other memory

```
while (TRUE) {
        tape_barrier();
        forall i,j {
                temp[i][j] = arr[i-1][j] + arr[i+1][j];
        }
        tape_barrier()
        forall i,j {
                arr[i][j] = temp[i][j];
        }
}
```

Figure 2: Red/black stencil.

model. The key point about LRC is that consistency information (invalidations) only moves with synchronization. The invalidate signal corresponding to a modification of a shared page only arrives at a process when that process synchronizes with respect to the process that performed the modification. Thus, new invalidations can be expected to arrive with synchronization.

In all cases, we rely on the underlying protocol layer to ensure correctness, regardless of when data arrives. Section 4 describes the demands that this requirement places on the underlying protocol.

3.1 Record-Replay Barriers

The most simple way in which we expect tapes to be used is in *recording* data movement in the first iteration of an iterative scientific application and *replaying* it in future iterations. Much of the remote latency can be hidden by sending the data before it is needed. Figure 2 shows pseudo-code for a simple grid application. Each process iteratively computes new values for all of the elements that it owns, using barriers and a temporary array to synchronize the read and write accesses to the shared array.

The only difference between this code and code written for a non-Tapeworm system are that the barrier calls are to specialized versions, rather than to the generic `cvm_barrier()`.

Pseudo-code for each process's barrier routine is shown in Figure 3. The purpose is to selectively send updates to remote processes before they are requested. Each process identifies data to be flushed to other processes by crossing the set of locally-created modifications with the set of data requested by other processes, and assuming that sharing patterns are static.

Each process records locally-created modifications, as well as data requests from other processes. This allows a process to directly track the data that will be needed by other processes during the next iteration. Tracking writes allows a process to identify new local modifications. Crossing such requests with the tape of local modifications allows us to create descriptions of the data that needs to be sent to other processes.

```
Tape            reqTape;        /* records requests from other procs */
Tape            writeTape;      /* records local writes */
Extent          reqExtents[NUM_PROCESSES];

tape_barrier()
{
    writeTape.stop_writing();
    reqTape.stop_reading();

    for proc in (all processes) {
        reqExtents[proc] += reqTape.flatten(proc);

        Tape *out = writeTape + reqExtents[proc];
        if (!out->empty()) {
            create message, add tape, and send to proc
        }
    }

    barrier();

    reqTape.reset();        reqTape.start_reading();
    writeTape.reset();      writeTape.start_writing();
}
```

Figure 3: Record/replay implementation

In more detail, each process uses `writeTape` to record local writes and `reqTape` to record requests during any single iteration. The `reqExtents[]` array is used to hold the set of all pages that each process has ever requested. The barrier procedure starts by flattening the remote request tape to sets of pages requested by each remote process. Each such set is unioned with the set of all previous pages requested by that process. The tape of local writes is then crossed with each such set to create a new tape naming the set of modifications that needs to be flushed to the corresponding process. C++'s operator overloading allows addition of a tape and an extent to be interpreted as "create a copy of the input tape, such that only pages described in the extent are included." For each such tape that is non-empty, a message is created, populated with the tape, and sent to the corresponding process.

This code assumes static access behavior. Applications with dynamic sharing patterns will only benefit to the extent that there is overlap between the sets of data accessed by consecutive iterations. Record/replay barriers for dynamic sharing patterns would only maintain extent information about recent iterations, rather than about all as in the static case.

3.2 Update locks

Update locks are modifications of the globally exclusive locks common to many parallel programming environments. Update locks use tapes and extents to combine data movement with synchronization transfers. Rather than using separate protocol transactions for synchronization and for data, update locks attempt to piggyback the data movement on top of existing synchronization messages. Tapes and extents are used to

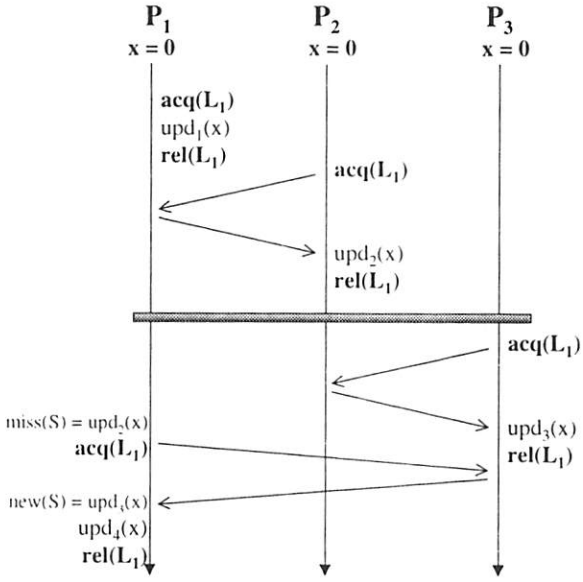


Figure 4: Auto-locks

identify and communicate the updates that are needed to validate shared data.

Auto-locks

Auto-locks attempt to exploit static access patterns by using past behavior to predict and eliminate memory faults during later lock synchronizations. The assumption is that the set of pages accessed during the $n+1^{th}$ acquire of any lock is similar or identical to the set of pages accessed during the n^{th} acquire of the same lock. Hence, we can avoid remote faults by ensuring that the pages accessed during the last lock acquisition (of the same lock) are valid when the lock acquisition is accomplished.

There are two sets of updates that need to be retrieved in order to prevent these remote faults. Let S be the set of pages that the requestor will access while holding the lock. This is the set of pages that the auto-lock mechanism will attempt to validate. The necessary updates can be divided into `miss(S)` and `new(S)`. `Miss(S)` consists of updates known (but not present locally) before the lock grant returns, whereas `new(S)` consists of new updates learned about from information piggybacked on the lock grant. The former set is empty if the pages in S are all valid when the lock acquisition begins.

Consider the example in Figure 4. For the sake of simplicity, assume that x is a single page. Prior to performing its second lock acquisition, P_1's copy of page x is invalid because the preceding barrier disseminated an

```
Tape        writes;
Extent      lockExtent;

/* executed by prospective holder of the lock */
void Tapeworm::lock_entry(int id)
{
        writes.reset(); writes.start_writing();
}

void Tapeworm::add_to_lock_request(Msg *msg, int id)
{
        Tape *empty = tape->missing_data_tape(lockExtent);
        msg->add(tape, (char *)empty, empty->size());
        msg->add(type_extent, (char *) lockExtent, lockExtent ->size());
}

void Tapeworm::read_from_lock_grant(Msg *msg, int id)
{
        read_data(msg);
}

void Tapeworm::lock_release(int id)
{
        writes.end_writing();
        lockExtent = writes.flatten();
}

/* executed by last holder of the lock */
void Tapeworm::add_to_lock_grant(Msg *msg, int pid)
{
        Tape            outTape;

        outTape = *(Tape *)msg->retrieve(type_tape)) {
        if (extent = msg->retrieve(type_extent)) {
                iMan->new_interval();
                outTape += get_new_tape(pid, extent);
        }
        outTape.add_data(msg);
}
```

Figure 5: Auto Lock Implementation

invalidation resulting from P_2's update. P_1's `miss(S)` therefore consists of $\texttt{upd}_2\texttt{(x)}$.

The `new(S)` set comes about because weakly-consistent protocol implementations often append consistency information to existing synchronization communication. In Figure 4, the lock grant at P_1's second lock acquisition returns knowledge of a third update, $\texttt{upd}_3\texttt{(x)}$. Hence, this latter update constitutes `new(s)` at the lock grant. All of the updates in either set are needed in order to validate the pages in S.

Figure 5 shows the code used to implement auto-locks in Tapeworm, lacking only comments and error-checking code. Each of the five routines is an upcall from the underlying implementation into the protocol code. The first four execute on the requestor's side, the last is executed by the previous holder of the lock. Tapeworm is implemented as part of a tapes protocol that specializes the default multi-writer LRC protocol. Therefore, all upcalls from CVM first call the Tapeworm routines, and then fall through to the corre-

sponding LRC routines that maintain memory consistency.

The data structures consists of `writes`, a tape used to record local modifications to shared memory, and `lockExtent`, an extent used to remember the set of pages accessed the last time the lock was held. The code starts recording modifications in `lock_entry()`, and stops in `lock_release()`. The `add_to_lock_request()` routine is called just before the lock request messages are sent. The auto lock routine adds an extent and a tape to this message. The extent is derived from the `writes` tape created during the previous lock access. The tape, created by `missing_data_tape()`, names all updates needed in order to validate the region covered by the extent. In other words, if page x of the extent's region is currently invalid, the tape specifies all updates that need to be applied to x in order to re-validate it.

The routine `add_to_lock_grant()` is called by the lock granter. This routine first retrieves `miss(S)` from the message and then creates `new(S)` by `get_new_tape()` to the extent sent in the request. These two tapes are added together, potentially resulting in a tape that includes modifications from several different processes. Finally, `add_data()` is used to load the tape data into the reply.

Finally, the requesting process uses `read_data()` to read and apply all updates from a message. If all has gone well, `read_data()` will also re-validate the entire shared region named by `lockExtent`.

User Locks

The second type of update locks, *user locks*, replace the implicit arguments of auto-locks with explicit buffer and length arguments. User locks are useful when the shared data accessed while a specific lock is held will change in some well-known manner. The interface to user locks include a simple buffer pointer and length. These parameters allow the program to specify a single contiguous section of shared memory that is likely to be accessed while the lock is held. Inside the lock operator, the region is converted to an extent, which provides an efficient and portable representation of the set of pages covered by the region.

This extent is used to create `miss(S)` as above. It is also appended to the lock request in order to identify `new(S)`. These quantities are handled similarly to the corresponding quantities in auto-locks.

User locks might be less accurate in anticipating data accesses than auto-locks. Programmers are often inaccurate, and locks may guard accesses to non-contiguous regions of shared data. Auto-locks can also accommodate slowly changing access patterns by using only recent data to inform subsequent lock requests.

```
start_produce()
{
    tape.reset();    tape.start_recording();
}

end_produce()
{
    tape.stop_recording();
    queue.add(tape);
}

Tapeworm::page_request (int pg_id, Msg *msg)
{
    if (Tape *tape = queue.search(pg_id)) {
        tape->add_data(msg);
    }
}
```

Figure 6: Producer-consumer regions

3.3 Producer-Consumer Regions

Many applications exhibit producer-consumer interactions. In these applications, one process *produces* a region of memory that is *consumed* by another process at an arbitrary later time. These types of communication are difficult to anticipate because the producer-consumer connections are often dynamic and can have low locality. If such regions are multiple pages, the consumer usually must fetch updates to each page separately, as the pages are accessed.

Tapes and extents can be used to aggregate these transfers by recording writes at the producer end, flattening the resulting tape to an extent, and storing it with the region pointer. When a process subsequently consumes the data by removing the pointer from the central repository, it also retrieves the corresponding extent.

Figure 6 shows the implementation of producer-consumer regions in Tapeworm. The application registers the region by bracketing its writes with `start_produce()` and `end_produce()` calls. In addition to stopping the recording, the latter enters the resulting tape into an ordinary queue. CVM first vectors page fault requests to the tape protocol, providing an opportunity to search the queue for a tape that contains the requested page. If the page is found, the entire region's data is appended to the reply message. While the total data transferred is the same as if the pages were transferred one at a time, the benefits of aggregating multiple requests into one can be significant.

Interface	Type	Description
*Tape *missing_data_tape(Extent *)*	c	Return tape describing updates needed to validate extent.
*Tape *get_new_tape(Extent *)*	c	Return tape describing "new" updates (see Section 3.2).
Tape::(start|stop)_(reading|requesting|writing)()*	c	Record accesses to shared memory.
*Extent *Tape::flatten()*	c	Return an extent describing pages mentioned by the tape.
*Tape::add_data(Msg *),* *Tape::read_data(Msg *)*	c, m	Add updates described by tape to message, read any such updates from message and incorporate into shared data.
*msg->add(msg_type, char *, int)* *msg->retrieve(msg_type, char **, int *)*	m	Allows arbitrary data to be added and retrieved from Msg objects.
Protocol::fault(int pg) *Protocol::page_request(Msg *, int)*	c, m	Upcalls to Tapeworm for local page faults and requests for local data from remote sites.
*Protocol::add_to_lock_request(Msg *, int)* *Protocol::add_to_lock_grant(Msg *, int)* *Protocol::read_from_lock_request(Msg *, int)* *Protocol::read_from_lock_grant(Msg *, int)*	m	Allows data to be piggybacked on top of existing synchronization messages.

Table 1: Low-Level Support for Tapes ('c' – consistency, 'm' – message)

4. Low-Level Support for Tape Protocols

Our tapes implementation is layered on top of CVM [1], a software DSM that supports multiple protocols and consistency models. CVM is written entirely as a user-level library and runs on most UNIX-like systems. CVM was created specifically as a platform for protocol experimentation.

New CVM consistency protocols are created by deriving classes from the base *Page* and *Protocol* classes. Only those methods that differ from the base class's methods need to be defined in the derived class. The underlying system calls protocol hooks before and after page faults, synchronization, and I/O events. Since many of the methods are inlined, the resulting system is able to perform within a few percent of a severely optimized commercial system running a similar protocol. Although CVM was designed to take advantage of generalized synchronization interfaces, as well as to use multi-threading for latency toleration, we use neither of these techniques in this study.

While tapes are conceptually independent of both the programming model and the particular protocol implementation, the underlying consistency protocol and system architecture must provide some basic support. Table 1 summarizes the required interfaces to the consistency mechanism and to the messaging subsystem. Those rows marked with a 'c' are requirements specific to the consistency protocol itself. Those marked with an 'm' are other hooks for creating and handling communication, or message, events.

4.1 Interactions with the consistency protocol

Tapeworm is layered on top of a consistency protocol, it is not a consistency protocol itself. In CVM's implementation, Tapeworm is a subclass of LmwProtocol, which is derived from the base Protocol class. LmwProtocol is the base multi-writer LRC protocol used by both CVM and TreadMarks [6]. All protocol calls other than those listed in the last two rows of Table 1 are passed directly through to LmwProtocol. Those in the last two rows are handled first by Tapeworm, and then passed down to the lower level. Porting Tapeworm to another protocol would require changing the base class, and re-implementing the functions in the first two rows of Table 1.

`Missing_data_tape()` and `get_new_tape()` are functions provided by the underlying protocol to Tapeworm, and were discussed in Section 3. The primary requirement that they impose on the underlying protocol is a *versioning* capability. This is the ability to generate update summaries, to apply them at remote sites, and to ensure that consistency is not violated throughout. In LRC, shared updates are summarized as *diffs* [8], and can easily be added to messages and applied at remote sites. Consistency correctness is preserved because diffs carry enough consistency information to determine when and where they should be applied.

This requirement can be problematic in the case of single-writer protocols like sequentially consistent page-based protocols [9]. Such protocols provide no way to determine whether a given copy of an object is current or not. However, single-writer protocols with support for object versions, i.e. the single-writer LRC protocol [1], provide the necessary basic mechanisms.

The next set of routines start and stop recording of various types of accesses to shared memory. The implementation of these routines is dependent on the underlying protocol. In the case of CVM's multi-writer protocol, *request* accesses are recorded by maintaining a log of data requests during the recorded interval. *Read* accesses are recorded by read faults during the recorded interval. Note that this approach does not necessarily capture every page that is accessed because faults do

App.	Input Set	APIs Used	Improvement			
			Speedup	Msgs	Misses	Bytes
Water	5 iters, 512 mols	lock, bar	14%	42%	83%	0%
TSP	18 cities	lock	7%	79%	94%	9%
Spatial	5 iters, 1024 mols	lock, bar	41%	96%	100%	15%
QS	$1x10^6$	lock, p-c	49%	53%	88%	0%
Gauss	1024 x 1024	flush	25%	67%	100%	2%
Barnes	8192 bodies	bar	40%	75%	48%	-2%

Table 2: Application Summary

not occur for valid pages. Additionally, this information is maintained with page granularity. *Write* accesses are recorded by maintaining a record of newly created *diffs*[8]. A diff is a data structure used by the underlying system to summarize modifications to a specific page.

`Tape::flatten()` returns an extent that lists the pages named by the accesses in the corresponding tape. `Tape::add_data()` and `Tape::read_data()` are functions provided by the underlying protocol to Tapeworm. They allow Tapeworm to copy the data named by a tape into or out of network messages.

`Protocol::fault()` and `Protocol::page_request()` are upcalls from the DSM to a protocol, in this case Tapeworm. Tapeworm specializes these calls to track accesses to shared pages. `Protocol::fault()` is called at local accesses to pages with the wrong permissions, i.e. reading an invalid page or writing a page without permission. Tapeworm uses this call to track reads and writes to shared pages. The `Protocol::page_request()` function is called when a remote site requests local data. This is used both for tracking requests (as with record/replay barriers), and for identifying and handling accesses by a consumer to producer/consumer regions.

4.2 Interactions with the message subsystem

Independent of the consistency protocol, Tapeworm must also have access to the messaging layer in order to add and retrieve data to existing messages, as well as to create Tapeworm-specific messages. The calls `msg->add()` and `msg->retrieve()` allow arbitrary data to be added and retrieved from CVM *Msg* objects. While *Msg* objects are specific to CVM, the same functionality could be made available without reference to specific message objects. However, this method would be less clear, so we have left the interface unchanged.

The last row of Table 1 shows upcalls from the DSM system to the consistency protocol. These are intercepted by CVM to provide hooks into existing messages. By adding data to these messages, Tapeworm can often avoid creating messages itself.

5. Performance evaluation

This section describes the performance of several applications, both with and without the use of Tapeworm's new synchronization primitives. Section 5.1 describes our experimental environment and Section 5.2 gives an overview of our application suite. The rest of the subsections describe the impact of Tapeworm on performance. Since each application was chosen to provide a different challenge to the synchronization library, we describe our results one application at a time rather than all at once.

5.1 Experimental environment

We ran our experiments over CVM's lazy multi-writer protocol on an eight-processor IBM SP-2. Each node is a 66.7 MHz POWER2 processor. The processors are connected by a 40 MByte/sec switch. The operating system is AIX 4.1.4.

CVM runs on UDP/IP over the switch. Lock acquires are implemented by sending a request message to the lock manager, which forwards the request on to the last requestor of the same lock. This takes either two or three messages, depending on whether the manager is also the last owner of the lock. Two-hop lock acquires take 779 µsecs, while three-hop lock acquires take 1185 µsecs. Simple page faults across the network require 1576 µsecs. Page fault times are highly dependent on the cost of mprotect calls (15 µsecs) and the cost of handling signals at the user level (120 µsecs). Minimal 8-processor barriers cost 1176 µsecs.

5.2 Application suite

Our application suite consists of one branch-and-bound lock application, TSP, one producer-consumer divide-and-conquer application, QS, two applications that combine both locks and barriers, Water (Water-Nsquared from SPLASH-2 [10]) and Spatial (Water-Spatial from SPLASH-2), one tree-structured barrier application, Barnes (also from SPLASH-2), and gauss (gaussian elimination with partial pivoting). While these applications are meant to be in some sense "representative," their more important common attribute was that each had characteristics that illustrate one or more facets of tape behavior. Note that there certainly exist applications for which tapes do not improve performance. Performance can even degrade if the access patterns assumed by the tape mechanisms called by an application do not match the actual sharing patterns in the application.

Protocol	Speedup	Remote Misses	Lock Pages	Updates Used	Comm KBytes	Messages				
						Lock	Barrier	Flush	Data	Total
Default	5.66	4852	0	-	6697	2786	196	0	4878	7860
Rec/Rep	5.78	3405	0	71%	6761	3016	196	420	3415	7047
User Locks	5.93	4336	1579	60%	6852	2642	196	0	4348	7186
User + Rec/Rep	6.14	1874	1550	64%	7736	2720	196	924	1950	5790
Auto-locks	6.16	3200	1566	70%	6683	2550	196	0	3200	5946
Auto + Rec/Rep	6.43	841	1535	68%	6655	2592	196	924	852	4564

Table 3: Water

Protocol	Speedup	Remote Misses	Lock Pages	Updates Used	Comm Kbytes	Messages			
						Lock	Barrier	Data	Total
Default	7.02	6058	0	-	6860	1124	28	6060	7212
User	7.22	4297	6161	88%	6648	1142	28	4272	5442
Auto-locks	7.48	387	6120	68%	6249	1134	28	387	1549

Table 4: TSP

Table 2 summarizes the maximum performance improvements on each of our applications. Details of the algorithms are deferred until the discussion of each application's performance. Overall, the best combination of options for each application eliminated an average of 85% of all remote page misses, 63% of all messages, and an average increase in speedup of 29%. For iterative programs, e.g. Barnes, Spatial, and Water, only the second and subsequent iterations were measured, in order to eliminate effects caused by the initial data distribution.

5.3 Application performance

Water

The first application is Water, an iterative molecular simulation. Water alternates phases in which locks are used and phases in which barriers are the only synchronization.

Table 3 shows the performance of Water with no tape optimizations, with record/replay barriers, with user locks, with automatic locks, and with both types of locks plus record/replay barriers. "Speedup" is relative to the single-processor time without CVM overhead. "Remote Misses" is the number of remote page faults incurred. "Lock Pages" is the number of pages that are re-validated by data moved as a result of one of the tape mechanisms. The "Updates Used" column shows the percentage of updates moved by the tape mechanism that are used at the destination. This column is omitted in some of the other application tables because it is near one hundred percent. "Comm KBytes" shows the total amount of data communicated during the measured portion of the application. Again, this column is omitted in some later tables because it is essentially unchanged across different runs. Finally, the last five columns show lock, barrier, flush, data (data request), and total messages.

Several trends are clear. First, auto-locks perform better than user locks. The reason is that user locks are difficult to specify statically. In at least one place, the region actually passed to the lock is only a guess at the data that will end up being modified.

Second, the sets of misses addressed by the lock and barrier mechanisms are disjoint: the number of misses eliminated with both mechanisms is almost exactly the sum of the misses eliminated by the mechanisms individually. Simple update protocols would perform similarly to the record-replay barriers, but be less effective at eliminating misses that are addressed by the update locks.

TSP

TSP is a branch-and-bound implementation of the traveling salesman problem. The central data structure is a global queue that contains partially completed tours. Processes alternately retrieve tours from the queue, split them into sub-tours, and put them back into the queue.

As shown in Table 4, TSP is almost exclusively lock-based. Locks are used to guard access to the central queue and to current minimum tour values. Barriers are used only during initialization and cleanup. We investigated both user locks and auto-locks. The results are shown in Table 4.

The first row shows the default TSP application. The second row shows performance with user locks. User locks are used to avoid misses when updating the "best" tour variable and when accessing the work queue. However, user locks can not specify the data that will be returned by a request for new work to perform, because the specific work has yet to be identified.

The auto-locks perform better because they retain a history of the last data that was accessed when the lock was held. This history is not an accurate predictor of

Protocol	Speedup	Remote Misses	Updates Created	Updates Used	Comm Kbytes	Messages				
						Lock	Barrier	Flush	Data	Total
Default	3.62	32677	4845	-	21727	764	518	0	65354	66636
Auto-locks	3.63	32494	4847	76%	21746	780	518	0	64988	66286
Rec/Rep	4.98	158	8950	98%	18924	762	518	1588	316	3184
Auto+Rec/Rep	5.12	11	8943	98%	18885	734	532	1589	22	2877

Table 5: Spatial

Protocol	Speedup	Remote Misses	Lock Pages	Messages				
				Lock	Barrier	Data	Tape	Total
Default	4.23	4499	185	3804	28	9110	0	12942
User	5.86	3377	1064	3830	28	6890	0	10748
User + PC	6.32	539	1563	3806	28	142	2096	6072

Table 6: QS

future accesses (witness the low "updates used" value), but is relatively complete.

Spatial

Spatial solves the same problem as Water, differing primarily in that the molecules are organized into three-dimensional "boxes." The sizes of the boxes are set so that molecules in one box interact only with molecules in neighboring boxes. The box structure allows synchronization and sharing to be done at the level of boxes rather than individual molecules, effectively aggregating much of the synchronization. This gain is partially offset by the overhead of maintaining the box structure.

Table 5 shows the performance of Spatial. The "Updates Created" column describes the number of separate per-page updates that are constructed by the underlying LRC system. The number of updates doubles with record-replay barriers because the default version is able to lazily create updates only at every other barrier.

Other than the overhead of creating and applying updates, this problem ends up having little impact on Spatial's performance. The multiple updates usually do not overlap, and therefore do not consume any more space or bandwidth than single updates. Second, few additional flush messages are sent because there are usually other updates destined for the same site. The messages would need to be sent even if the excess updates were not produced. The flush versions actually send less data than the non-flush versions because the large flush messages have less system overhead than individual update requests.

Auto-locks have little effect on Spatial's performance. The reason is that locks are used mainly to arbitrate access to the linked lists that tie molecules to boxes. The auto tape mechanism only prefetches the pages containing these pointers, not the pages containing the molecules themselves.

Nonetheless, the overall impact of the flush mechanism is to improve performance by over 41%.

QS

QS is a parallel implementation of QuickSort. Again, the central data structure is a global queue that contains partially computed values, which are iteratively removed, refined, split, and inserted back into the queue until all are complete. QS differs from TSP in that the chunks of data that are taken out of the queue are merely pointers to the actual data. Hence, we use the producer-consumer regions that were discussed in Section 3.3.

Table 6 shows three versions of the QS program, with statistics as for TSP. The only new statistic is the "tape" message type. The first row shows the default implementation. The second row shows the results of a run in which all accesses to the central queue are through user locks. The regions passed to the user locks are the entire centralized queue structure. As this structure is updated frequently, the user locks eliminate all misses on the pointer data structures, about one fourth of all remote misses.

The row labeled "User+PC" contains statistics reflecting the producer-consumer tape functions discussed in Section 3.3. The number of remote misses is reduced six-fold over the version with just user locks. The total number of messages is reduced by 53%, and speedup is increases by 49%.

Barnes

Barnes is the n-body galactic simulation from SPLASH-2, modified by Ram [11] to contain only barrier synchronization. Because of this modification, fine-grained tasks such as *make-tree* are now performed sequentially. This modification effectively increases the synchronization granularity.

Protocol	Speedup	Remote Misses	Updates Used	Comm KBytes	Messages			
					Barrier	Flush	Data	Total
Default	3.88	4177	-	15767	140	0	31826	31966
Rec/Rep	5.43	2157	87%	16047	140	576	7266	7982

Table 7: Barnes

Protocol	Speedup	Remote Misses	Updates Used	Comm KBytes	Messages			
					Barrier	Flush	Data	Total
Default	3.45	14294	-	32280	7160	0	14294	21454
Flush	4.31	0	100%	31673	7160	0	0	7160

Table 8: Gauss

Table 7 shows that Barnes differs from the other applications in that use of the tape mechanism is only able to eliminate about half of the remote misses. This is primarily because of a lack of locality across iterations. Processes access new pages during each iteration, and the system is therefore unable to anticipate all accesses. Nonetheless, 87% of updates flushed at barriers are eventually used, and total messages sent drops by a factor of four.

Gauss

Gauss is an implementation of Gaussian elimination with partial pivoting. Essentially, it consists of a 2-D grid, with rows assigned to processes in chunks. During each iteration, a new row is chosen as the "pivot". Each process updates all rows *after* the pivot row. The pivot row and column index need to be propagated to all other processes.

This method of updating plays havoc with standard update protocols. The problem is that each pivot is only flushed once, meaning that historical information can not be used to determine that the data needs to be broadcast. Application input is essential. We used tapes to build two new routines called "cvm_start_flush()" and "cvm_stop_flush()". These routines use a tape to record all shared modifications, and to broadcast them to all other processes.

Gauss's performance is shown in Table 8. All remote misses are eliminated. However, overall speedup is still mediocre because the last iterations have too little computation to make parallelism worthwhile.

5.4 Discussion

The tape mechanism's advantages are performance and simplicity. In evaluating performance, we distinguish between the performance of the tape layer itself, the performance of Tapeworm, the specific synchronization library discussed in this paper, and the potential performance improvements of other synchronization libraries that could be built using tapes.

The tape layer itself adds very little overhead. Recording page reads and writes adds only a few instructions to the page fault handlers. The runtime cost of manipulating tapes and extents is also small. Extents are implemented as bitmaps in our current prototype. They are therefore fast, but reasonably expensive in terms of memory consumption. Since the constituent elements of extents are pages, the size of an extent is proportional to the number of shared pages. Currently, the largest applications we run share approximately thirty-two megabytes. Assuming 8k pages, this results in a bitmap of 512 bytes. On the other hand, water uses less than 500k of data, resulting in bitmaps of only eight bytes. If the current representation becomes unacceptable, extents could be implemented as sets of bitmaps, and would have size proportional to the working set of pages. Tapes are currently implemented as sequential records of events, and are therefore of size proportional to the number of recorded events. Similar to extents, more sophisticated representations for tapes are possible in the event that their size or runtime cost grows too large.

As far as the effectiveness of the specific synchronization library discussed in this section, Table 2 shows that Tapeworm eliminates an average of 85% of all remote access misses. The percentage of access misses eliminated can be termed the coverage of the protocol. The accuracy of the protocol can be characterized by the number of updates sent but not used. These updates are pure overhead, but do not affect correctness. This quantity is given by the "Updates Used" column in Table 3 through Table 7. Tapeworm's average accuracy is 91%. Assuming a uniform distribution of diff sizes, this implies that the average bandwidth overhead is only nine percent. However, the number of extra messages is likely to be a much smaller percentage. Most of these extra updates are sent in messages that would have to exist for other updates or synchronization, even if the useless updates were not sent.

One last aspect of this effectiveness is whether Tapeworm results in a significant number of extra updates being created and applied. This occurs only in Spatial. However, it does not result in either extra messages or

data, so we conclude that the effect on Spatial's performance is negligible. This effect could be significant in other applications. We expect that specializing barriers, as described in Section 3.1, would minimize this effect.

Mechanisms such as auto-locks and record/replay barriers also incur overhead in that they need to be trained before being used. Faults incurred during the initial use of these mechanisms can be termed cold misses. Faults avoided during subsequent synchronizations are always conflict misses for our implementation because CVM relies on the underlying virtual memory system to handle capacity problems. All results presented in this paper represent the steady state execution of applications after the cold misses are complete. Assuming static sharing behavior, however, the percentage of potential faults that are cold misses can never be higher than *1/n*, where '*n*' is the number of iterations timed. Hence, cold misses are unlikely to be important for realistic runs.

The second claimed advantage of tapes, simplicity, has two parts: simplicity of use and simplicity of support. Our claim of simplicity for support is based on the amount of code needed to build the tape mechanism. The total size of the CVM system is about 15,000 lines of commented code, including debugging statements. The tapes support layer consists of less than 500 lines of C++ code, and the Tapeworm synchronization library is an additional 400 lines.

Finally, one possible critique of this work is that it is too closely tied to the multi-writer LRC protocol used to generate the above results. Actually, this protocol is not an ideal substrate. The reason is that the tape protocol relies on noticing all shared modifications that occur when recording. However, multi-writer LRC uses lazy diffing, meaning that once a process has started writing a page, the page remains writable until any of the modifications are requested elsewhere. Hence, a tape that starts recording writes *after* a page has already been made writable will not necessarily notice additional writes to the same page. This problem could be avoided by removing write permission from all pages when any recording of writes is started. This will result in more complete information (and possibly fewer residual remote misses), but incur higher overheads for the actual recording.

6. Related work

Record/replay barriers were first implemented by the Wind Tunnel project [12]. This work focused on providing support for irregular applications by coding application-specific protocols, one of which implemented a record/replay barrier. Later work in the same project resulted in a protocol-implementation language called Teapot [13]. This work is similar to ours in that both are trying to expose protocol handles to application or library builders. However, the Teapot language is more complex. More lines of Teapot code are required to implement a sequentially consistent invalidate protocol than the corresponding protocol written in C++ on CVM. Also, Teapot protocols perform both data movement and maintenance of correctness, whereas consistency can not be violated in any synchronization library built on top of tapes. One major advantage of Teapot is that it leverages existing cache protocol verifiers to automatically verify Teapot programs.

Our work has similarities to work performed at Rice University on compiler-DSM interfaces [14]. The `missing_data_type()` routine is essentially the information-gathering phase of the TreadMarks [6] `validate()`. Some of the update work we describe is similar in spirit to the TreadMarks `push()` command. However, our work not only provides ways to manipulate data, as with TreadMarks, but it also provides ways to gather this information dynamically through tapes. While the TreadMarks work assumes all information is provided by the compiler, our work provides a way for the user or synchronization library to gather this information at runtime. For instance, our tapes allow us to dynamically determine the extent of the data being accessed, while this information is assumed to be known by the compiler in TreadMarks. Our work also allows the user to manipulate discover and manipulate shared modifications at a high level. Recent work at Rice has investigated automatic determination of extent-like objects in shared memory applications [15].

Tapes are not a consistency mechanism, but they can be used to tune the interface of a CVM-like system so that it behaves as if a dissimilar underlying protocol were used. For example, our work could be used to build a synchronization interface that would closely approximate the data communication characteristics of Midway's [16] or CRL's [17] update protocol. While both systems differ from CVM in many ways, one of the key differences is that both Midway and CRL use update protocols. Unnecessary updates are avoided by limiting the updates to shared regions that are explicitly associated with synchronization. The auto-locks described in Section 3.2 would approximate these data movement patterns, modulo excess invalidations caused by false sharing.

Similarly, a tape is not a *scope,* but they can be used to build a synchronization interface that superficially mimics *scope consistency* (ScC) [18]. The two would differ in that ScC is a consistency model, whereas any interface built using tapes is merely a data movement mechanism that exists on top of the underlying consistency model. Hence, whatever claims are made as to the relative benefits of ScC and LRC as a consistency

model still apply. However, tapes can be used to greatly reduce communication traffic in either case. Since the existing ScC implementation is home-based, all updates are constrained to move through the home node. Therefore, data communication between processes P_1 and P_2 must involve the home nodes of any data communicated. The tapes-based approach can move less data, and certainly use fewer messages, than the home-based approach for all cases where the home nodes are not one of the communication endpoints. We plan to investigate the performance of a tapes layer on top of ScC in the future.

This paper has discussed the use of tapes to improve performance of software DSM systems, but it may also be relevant in the context of hardware shared memory systems. For instance, the prefetch and poststore primitives of the KSR-2 [19] implement user-initiated data movement on top of the underlying consistency protocols. Other work [20] generalized these primitives to allow the destination of pushes to be specified either by runtime copyset management or by specific calls initiated by application programs. By augmenting these primitives with the ability to read and store copyset information for future iterations, tapes could be supported on top of this type of system with only a minimal runtime layer. Even with an efficient implementation, however, such a system would probably only be useful with large cache lines, i.e. 128 or more bytes.

Shared memory systems with dedicated protocol processors [21, 22] might turn out to provide the best possible platform for tapes implementations. Tape code executing on the protocol processors could track data and synchronization accesses without ever involving the application processor.

7. Conclusions

This paper has described the tape mechanism, and its use in tailoring data movement to application semantics. Tape-based synchronization libraries are layered on top of existing consistency protocols and synchronization interfaces, meaning that incorrect choices (whether by heuristics or programmers) affect only performance, not correctness.

The tape mechanism is ideally suited to direct data movement because it allows shared accesses to be recorded, grouped, and manipulated at a very high level. These tapes can be used to predict future data accesses and to eliminate subsequent misses by moving data before it is needed.

We used the tape mechanism to build Tapeworm, a new synchronization library that uses information gathered at runtime to reduce access misses. Tapeworm's interface consists of auto-locks, producer-consumer regions, and record/reply barriers. Auto-locks pre-validate data that is accessed while locks are held. Producer-consumer regions use the first access to a region as a hint to request the rest of the region before it is needed. Record/replay barriers allow accesses to be recorded during one iteration and then played back during future iterations. The combination of these mechanisms allows Tapeworm to eliminate an average of 85% of remote misses for our applications, 63% of all messages, and to improve overall performance by an average of 29%.

Tapes have at least two major advantages over optimizations of specific protocols. First, tapes provide a high-level abstraction of shared accesses, and are protocol-independent. Tapes make few requirements on the underlying protocol, providing a terse, powerful approach to managing data movement. Second, tape mechanisms can be implemented and used incrementally. Applications can be completely debugged before any tape mechanisms are added. One by one, tape mechanisms can be used to improve data movement at inefficient points in application executions.

This work is complementary to recent work in parallelizing compilers [5, 14]. Tapes improve performance by exploiting repetitive access patterns. Identifying such patterns *with high degree of probability* in the compiler is much easier than generating explicit message-passing code for the data movement.

We conclude that the tape mechanism is a promising approach to creating high-performance synchronization libraries. Future work will investigate more sophisticated automatic interfaces, and the use of tapes in creating debugging libraries [23].

8. References

[1] P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," in *Proceedings of the 16th International Conference on Distributed Computing Systems*, 1996.

[2] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[3] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[4] Y. Zhou, L. Iftode, and K. Li, "Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory

Systems," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, October, 1996.

[5] C.-W. Tseng and P. Keleher, "Enhancing Software DSM for Compiler-Parallelized Applications," in *11th International Parallel Processing Symposium*, 1997.

[6] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, pp. 18--28, February 1996.

[7] L. D. Wittie, G. Hermannsson, and A. Li, "Eager Sharing for Efficient Massive Parallelism," in *Proceedings of the 1992 International Conference on Parallel Processing (ICPP '92)*, August 1992.

[8] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[9] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," in *Proceedings of the 1988 International Conference on Parallel Processing*, August 1988.

[10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[11] R. Rajamony and A. L. Cox, "Performance Debugging Shared Memory Parallel Programs Using Run-Time Dependency Analysis," in *Proceedings of Sigmetrics '97*, June 1997.

[12] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. H. J. R. Larus, A. Rogers, and D. A. Wood, "Application-Specific Protocols for User-Level Shared Memory," in *Supercomputing 94*, 1994.

[13] S. Chandra, B. Richards, and J. R. Larus, "Teapot: Language Support for Writing Memory Coherence Protocols," in *SIGPLAN Conference on Programming Languages Design and Implementation*, 1996.

[14] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System," in *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems*, 1996.

[15] C. Amza, A. L. Cox, K. Rajamani, and W. Zwaenepoel, "Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory," in *Proceedings of the Principles and Practice of Parallel Programming*, 1997.

[16] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The Midway Distributed Shared Memory System," in *Proceedings of the '93 CompCon Conference*, February 1993.

[17] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach, "CRL: High-Performance All-Software Distributed Shared Memory," in *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, 1995.

[18] L. Iftode, J. P. Singh, and K. Li, "Scope Consistency: a Bridge between Release Consistency and Entry Consistency," in *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1996.

[19] "Kendall Square Research. Technical Summary," 1992.

[20] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak, "Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors," in *Supercomputing*, 1995.

[21] J. Kuskin et al., "The Stanford FLASH Multiprocessor," in *Proceedings of the 21th Annual Internationa Symposium on Computer Architecture*, April 1994.

[22] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and Typhoon: User-Level Shared Memory," in *Proceedings of the 21th Annual International Symposium on Computer Architecture*, April 1994.

[23] D. Perkovic and P. Keleher, "Online Data-Race Detection via Coherency Guarantees," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996.

MultiView and Millipage – Fine-Grain Sharing in Page-Based DSMs

Ayal Itzkovitz Assaf Schuster
Computer Science Department
Technion – Israel Institute of Technology
{ayali,assaf}@cs.technion.ac.il

Abstract

In this paper we develop a novel technique, called MULTIVIEW, which enables implementation of page-based fine-grain DSMs. We show how the traditional techniques for implementing page-based DSMs can be extended to control the sharing granularity in a flexible way, even when the size of the sharing unit varies, and is smaller than the operating system's page size. The run-time overhead imposed in the proposed technique is negligible.

We present a DSM system, called MILLIPAGE, which builds upon MULTIVIEW in order to support sharing in variable-size units. MILLIPAGE efficiently implements Sequential Consistency and shows comparable (sometimes superior) performance to related systems which use relaxed consistency models. It uses standard user-level operating system API and requires no compiler intervention, page twinning, diffs, code instrumentation, or sophisticated protocols. The resulting system is a "thin" software layer consisting mainly of a simple, "clean" protocol that handles page-faults.

1 Introduction

The basic mechanism for implementing software distributed shared memory systems (DSMs) was described for the first time in a seminal paper by Li and Hudak [15] and implemented in the Ivy system [14]. The method relies heavily on the operating system's virtual memory page protection mechanisms, enforcing a sharing granularity which is equal to the size of the virtual memory page (*page-based* DSMs). Page sizes, typically a few kilobytes, are usually much larger than the actual sharing granularity of the applications. Therefore, the main problem that researchers have faced in developing page-based DSMs has been *false sharing*, where two or more hosts use different variables that happen to reside in the same page. False sharing can cause a severe performance degradation of programs running on software DSMs and may even lead to slowdowns.

There have been many attempts to overcome the false sharing problem. An extensive study has been conducted and numerous works on relaxing the memory consistency have been written, including [1, 3, 5, 7, 9, 12, 22, 26], to mention only a few. Relaxing the consistency enhances parallelism and may significantly reduce the required communication for memory synchronization. Memory synchronization is generally controlled by calls to a synchronization primitive or a method which is associated with one. Even when not much work is involved, relaxed consistency models do require that the programmer modify the code and be aware of the semantics of the memory behavior. As a result, DSMs using relaxed consistency models trade the abstraction of the underlying memory system for added efficiency.

A different approach was proposed in the Blizzard and the Shasta systems [18, 19, 20]. In order to circumvent the false-sharing problem and provide fain-grain access, Shasta avoids using the virtual memory protection mechanism. Rather, it instruments binary code, wrapping loads and stores with instructions to check for the availability of the data and to maintain its consistency. The result is a fine-grained DSM, capable of sharing memory blocks of arbitrary size. However, high overhead is introduced by the wrapping instructions, which necessitates aggressive optimization techniques.

In this paper we propose a new method, called MULTIVIEW, which allows the efficient implementation of fine-grained DSM. Although MULTIVIEW does use the virtual memory protection mechanism, it is capable of manipulating the memory in variable size blocks, called *minipages*, which are smaller than the virtual memory page size. MULTIVIEW involves

little overhead; it usually requires no modifications by the programmer, nor does it require post compilation or code instrumentation of any kind.

MULTIVIEW provides two notable advantages. First, false sharing can be avoided simply by associating variables with individual minipages. The system then manages sharing of program variables rather than full pages. Second, MULTIVIEW enables a DSM implementation that completely avoids buffer copying in the DSM layer. For this reason MULTIVIEW is well suited for integration with high performance messaging layers like Active Messages [24], FastMessages [16] and the VIA interface.

We have implemented a system named MILLIPAGE - a high-performance fine-granularity page-based DSM. MILLIPAGE uses MULTIVIEW both for achieving fine granularity and for enhancing performance. Despite the fact that it uses the virtual memory page protection mechanism (and thus can be viewed as "page-based"), MILLIPAGE supports sharing of memory at any granularity. Furthermore, sharing in small granularity imposes only a negligible overhead in MILLIPAGE.

It has recently been noted, e.g. in [23], that the latest advances in communication speed make the complexity of the underlying DSM protocols a non negligible factor in the overall system performance. A notable aspect of MILLIPAGE is its efficient support of Sequential Consistency with a very simple and "clean" protocol, which leads us to the notion of a *thin-layer* DSM. The key element in thin-layer DSMs is the simplicity of handling a request for shared data. There is no need for page twinning, which consumes memory, nor for diff operations, which occupy the cpu, code instrumentation, which blows up the instruction count, or sophisticated protocols which complicate the system. As a result, thin-layer DSMs are simple to develop and debug, easy to use, and impose little overhead on the local operating system and the communication network, beyond that which is required by the applications.

It was recently shown that reducing the granularity in systems which implement strict consistency may achieve performance comparable to that of systems implementing relaxed consistency memory models [19, 27]. In accordance with these findings, Sequential Consistency was employed in MILLIPAGE: initial performance evaluation shows results comparable or superior to those obtained in systems which employ relaxed consistency models.

MILLIPAGE is fully operational in the Distributed Systems Laboratory at the Technion - Israel Institute of Technology [1]. MILLIPAGE uses the Illinois FastMessages [16] on a cluster of 8 Compaq 300Mhz Pentium II machines, interconnected by a Myrinet switch, and running Microsoft Windows-NT.

The rest of this paper is organized as follows. The following section describes the MULTIVIEW technique and how to generate and control minipages. In Section 3 we describe the design of MILLIPAGE; we discuss important issues that affected the DSM architecture, issues which arise from applying the MULTIVIEW technique and the integration of fast messaging libraries. Initial performance evaluation of MILLIPAGE is provided in Section 4. Finally, we discuss future work and open research topics.

2 The MultiView Technique

Unlike object-based DSM systems, which require a specially tailored compiler or binary code instrumentation, page-based DSM systems provide a transparent and portable software layer that can be developed and used on standard platforms. The main disadvantage of page-based DSMs is that the smallest unit of sharing (the granularity) is the page size, which is determined by the operating system and the underlying processor architecture. The size of a page, as large as 4KB for the Intel Pentium and 8KB for the Digital Alpha, is three orders of magnitude larger than the memory addressing granularity. In this section we propose a novel technique, called MULTIVIEW, which enables the sharing of memory in fine granularity as small as that of the memory addressing unit.

2.1 The Basic Idea

Consider three variables x, y, and z, whose size is smaller than that of a page. In a page-based DSM system, the memory for these variables may be allocated on the same page, resulting in false sharing. Consider a mapping of the page that contains these variables to three different, non-overlapping, virtual address regions, starting at addresses v_1, v_2, and v_3, so that each of the locations in the page can be viewed via three different virtual addresses. Each of the regions is called a *a view*. Since access permission is controlled through the virtual memory mechanism, it follows that protection and fault handling can now be applied in three different and independent ways, one for each view. Consequently, the mechanism for maintaining page consistency can

[1] Updated information can be found in our web site at http://www.cs.technion.ac.il/Labs/Millipede

also arbitrate between three different policies, each using the access capability setting of one of the views. Figure 1 depicts this situation.

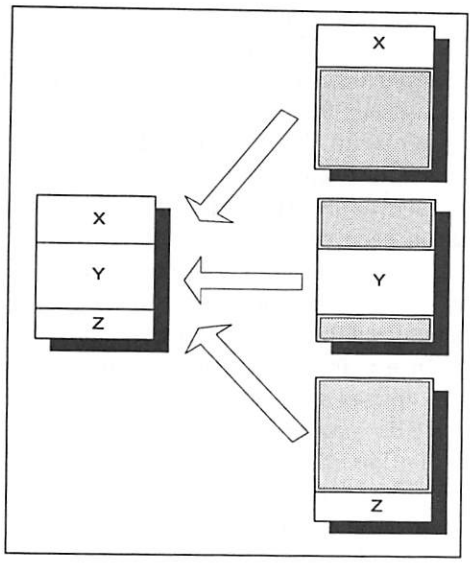


Figure 1: Mapping three virtual views to a single memory page in which three variables reside.

Now suppose the application uses different views to access different variables, say, the variable x is accessed via $v_1 + \text{offset}(x)$, y via $v_2 + \text{offset}(y)$, and z via $v_3 + \text{offset}(z)$. Then, although the variables reside in successive memory addresses on the same physical page, they are seen by the application as if placed in three different (virtual) pages. It is now possible to manage a separate access policy for each variable using its respective view. Moreover, an independent consistency protocol can be implemented for each variable, despite the fact that all three reside on the same page.

2.2 Views, Vpages, and Minipages

Most DSM systems provide consistency guarantees for a single, large, contiguous region of shared addresses. In systems which implement MULTIVIEW, the shared space is mapped to several such regions of virtual addresses, called *views*. Consequently, each memory element can be accessed via the virtual memory mapping mechanisms, using any of the views. Since views must be managed in granularity of pages, each view consists of a sequence of virtual memory pages which we call *vpages*.

MULTIVIEW lets the application manage each memory element through a dedicated view, or a vpage, which is said to be *associated* with this element. Because each memory element (in the above example, a variable) can now be managed independently regardless of its size, we call it a *minipage*. Minipage sizes vary; they can be as large as the virtual page size or as small as the basic memory addressing unit.

A minipage is identified by the associated vpage number and a pair < offset, length > which indicates the region inside the vpage where the minipage resides. A protection is controlled for the minipage using the virtual memory mapping mechanisms by manipulating the associated vpage protection. A `NoAccess` protection indicates a non-present minipage, a `ReadOnly` protection is set for read copies, and a writable copy gets a `ReadWrite` protection. Copying minipages between the hosts, invalidating minipage copies, and changing access permissions are all done according to the consistency guarantees and the protocols implementing them.

The basic feature which enables the implementation of a fine granularity DSM using the MULTIVIEW method is the independent manipulation of access permissions for minipages which physically share the same memory page, but are accessed by the application via different vpages. For instance, the following protocol implements Sequential Consistency. When a read fault occurs, i.e., when the application attempts to read a minipage for which the associated vpage has a `NoAccess` protection, an accessible copy of that minipage is located in one of the hosts and brought in. Accessing the minipage is then enabled by changing the protection of the associated vpage to `ReadOnly`. Similarly, upon encountering a write fault, a copy of the minipage is retrieved, all other copies are invalidated, and a `ReadWrite` protection is set for the associated vpage.

2.3 Minipages and Views Layouts

Preparing the minipage layout can be done in both static and dynamic fashion. Static layout may divide each memory page into k minipages of equal size. This way, it is easy to calculate the minipage borders when a fault occurs. Static layout may therefore be appropriate for general purpose caching and global memory systems, in order to reduce the page size by a fixed factor [10]. In the dynamic layout each allocation in the shared memory defines its own minipage according to the allocation size, and this minipage is associated with its own vpage. The system should therefore store and maintain a minipage-table (MPT) with the appropriate < offset,length > pair specified for each minipage. Large allocations should still reside in a contiguous region of addresses.

MILLIPAGE design is based on the dynamic layout; it is this option on which we focus in the rest of this paper.

2.3.1 The Privileged View

Multithreaded DSMs commonly use *server threads* for DSM management, and *application threads* for carrying out computation. We call the views that are used by the application threads *application views.*

In addition to the fine granularity capabilities, MULTIVIEW enables another useful feature. An additional, separate view is constructed, called the *privileged view.* The protection of the privileged view is fixed and set to `ReadWrite`. The DSM server threads may use it at all times to access the memory, and are thus not constrained by the DSM memory protections, as determined by the consistency guarantees imposed on the application views.

There are two common DSM operations which highly benefit from using the privileged view. First, atomic minipage updates can now be performed in user-mode. While access is blocked through the application views, the DSM server thread can freely access minipages via the privileged view. Once the modification is complete, the protection in the application views can be reduced, thus enabling the application threads to access the modified memory. In this way multithreading can be employed safely.

Secondly, buffering and copying of long messages can be avoided. The privileged view may be used to directly send/receive minipages from/to the user space. While communication activities are taking place using the privileged view, application thread access to the same memory region is forbidden by the protection imposed on the application views. In this way, buffer copying is completely eliminated in the DSM layer. Later, in Section 3.5, we describe how all this is implemented in MILLIPAGE and integrated with the Illinois FastMessages.

2.4 Implementation Issues

We have implemented MULTIVIEW in Windows-NT in a DSM system called MILLIPAGE. Mapping several views of virtual addresses to a shared memory region was accomplished using the *file mapping* API, as follows. First, a *memory object*[2] is created by calling the `CreateFileMapping` API. A memory object is simply a virtual memory region, allocated in the kernel address space of a process, and backed-up by the paging file. This memory object is the shared region on which minipages will be allocated.

Suppose the maximal number of minipages that reside on the same page of the memory object is n. We thus need $n+1$ different views of the memory object: n application views for use by the application threads and one for the privileged view. The views are created using the `MapViewOfFile` API, which is called once for each established view.

For each application, MILLIPAGE keeps a single process on each of the hosts. By carefully configuring the DSM addresses, the views are guaranteed to map to the same addresses in all processes, so there is no need for address translations between the hosts. Once mapped, the protection of vpages (recall that vpages are the virtual pages composing a view) can be manipulated independent of the protection of other vpages that are mapped to the same memory object page. One of the views is made privileged with the protection of all its vpages set to `ReadWrite`. This view is used by the DSM server threads to read and update the memory object.

The construction of the views is performed during system initialization. When the application issues an allocation request, the DSM searches for a suitable region in the memory object, and defines it as a minipage (or a set of consecutive minipages). The DSM associates the newly defined minipage with one of the application views. More precisely, suppose an allocation procedure defines a new minipage $\mathcal{M}$. $\mathcal{M}$ is associated with a certain vpage in one of the views, and an address p in this vpage is returned. During allocation, a new entry is formed in the minipage-table MPT, containing both the offset of p in the vpage and the size of $\mathcal{M}$.

If mapping to $\mathcal{M}$ spans several vpages in the associated view, the above is generalized in a straightforward way.

Figure 2 describes the views configuration and the dynamic allocation of minipages to variables during `malloc` calls.

3 System Design

3.1 Design Goals

MILLIPAGE is a software-only implementation of a fine-granularity strictly-consistent distributed shared memory. Although MILLIPAGE can be seen as a page-based DSM, it is not limited to sharing memory in granularity of full pages.

The main design goals of MILLIPAGE are outlined below. We discuss them throughout this section.

[2] A memory object is called a *memory section* in Windows-NT terminology [21].

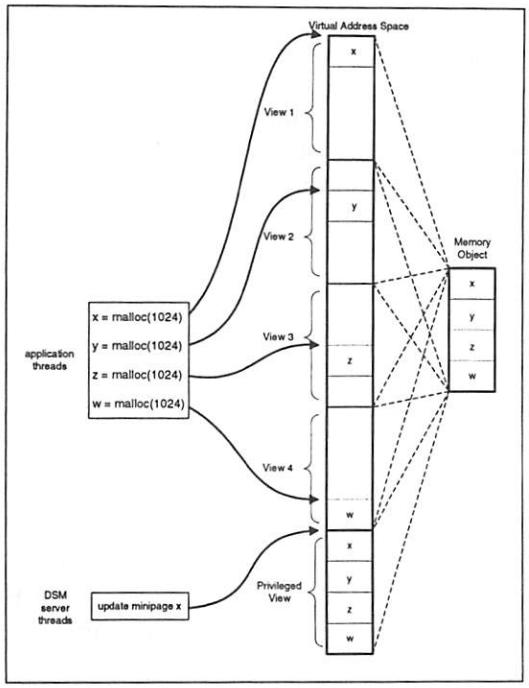


Figure 2: Views configuration and dynamic minipage allocation for independent variables. DSM service and control threads use the privileged view. In general, a memory object may be larger than a page and there may be "gaps" between the views.

- User level software implementation.
- Efficient strict memory model.
- Fine granularity DSM.
- Thin layer DSM.
- Integration with fast networking media and fast messaging packages.
- Multithreaded support to efficiently utilize SMP machines.

3.2 Shared Memory Model

The programming model in MILLIPAGE is Sequential Consistency, which is implemented through the Single-Writer/Multiple-Readers (SW/MR) protocol: at any point in time, for any minipage, there can be either read copies or a single writable copy. Thus, parallel applications run on MILLIPAGE as if they were executing on a physically-shared memory SMP machine; there is no need for either explicit or implicit memory synchronization.

Aside from the initial setting of the maximal number of views and the size of the shared-memory, the allocation process is transparent from the programmer point of view. Some restrictions do apply, however, in the sense that only dynamic allocations can be shared; those are the only allocations managed by the DSM mechanism. For this reason, the application should be written so that the sharing unit is equal to the allocation size.

Allocating from the shared memory is performed via a `malloc`-like API. The returned pointer can point to any of the application view's address spaces (but never to the privileged one). It can then be used in the usual way as if it had been returned by the standard `malloc` call.

3.3 Protocols

An important goal in the design of MILLIPAGE was to encapsulate the DSM functionality in a very thin software layer. This was accomplished by implementing a very simple SW/MR protocol, which we now proceed to describe.

On each host a single MILLIPAGE process is started, running both application and server threads. One of the processes is elected as the *manager*. As part of the manager role, it is in charge of maintaining the directory information of minipage and minipage copy locations, minipage sizes, and the association of view addresses with their minipages. This information is stored in the minipage-table (MPT), which is stored at the manager host.

All requests for missing minipages (resulting from a fault) are sent to the manager, which redirects them to the appropriate hosts. Requests which arrive while an earlier request to the same minipage is still in process are queued in the manager.

A request which arrives at the manager contains only the faulting address. The manager looks it up in the MPT and stores the translation information (the minipage base address, its size, and its address in the privileged view) in the message header where the appropriate space has been reserved. When the message is forwarded, it carries the translation information.

The manager-centered design significantly simplifies the DSM layer for the non-manager processes. Whenever a fault occurs, the fault handler issues a request and sends it directly to the manager. No computation or local search in any data structure is required. The thread then waits on an event while its request is serviced.

When the reply arrives, it is handled by a DSM server thread, which receives the message in two stages: first, the message header arrives, containing

the original request and the translation information. Next, the minipage contents are received directly at the appropriate address in the privileged view, as specified in the request header. When the receive operation completes, the protection for the minipage is set and the faulting thread is signaled to continue its execution.

The manager's role is essentially to mark and forward requests to hosts, and to maintain the MPT. If a read copy is requested, the manager updates the minipage copyset and forwards the request. If an exclusive write copy is requested, the manager first chooses one of the hosts in the copyset, instructs all others to invalidate their copies, and then forwards the request to the remaining one. This host will then invalidate its copy and send the minipage directly to the host where the fault occurred. The pseudo-code of the *complete protocol* is given in Figure 3.

Once a fault is served and the faulting thread wakes up, it sends an additional ack message to the manager. Although this additional message might seem to reduce performance, it actually solves a few potential problems. First, a possible livelock caused by race conditions on two or more threads is eliminated. This is reminiscent of the delta mechanism [6], which ensures that a page remains in the host for a certain amount of time before its removal is permitted. Second, the potential need for message queuing in the non-manager hosts is eliminated. A host which receives a request is never in the process of acquiring the same minipage, nor has it given the minipage away. Hence, a request which arrives at a non-manager host can always be served immediately, completely eliminating the need for buffers.

Since all the messages which are sent to and by the manager are small (32 bytes in our current implementation), reading and writing them to and from the network does not involve much overhead, leaving the manager highly responsive.

3.4 The MILLIPAGE Library

MILLIPAGE is implemented as a win32 library on Windows-NT. It exports several APIs for the use of application development. Its interface includes an initialization routine, spawning local and remote threads, a memory allocation routine, and common synchronization calls such as barriers and locks.

Applications can be compiled with any standard compiler. They should then be linked with the MILLIPAGE and the FM libraries. The final executable integrates the DSM run-time system with the application code and can be started concurrently on several hosts. Figure 4 shows the process of preparing an application to run on top of MILLIPAGE. Since MILLIPAGE is multithreaded and its architecture supports multithreaded applications, only a single instance of the application should be executed on each host, even if this host is a multi-processor (SMP) machine.

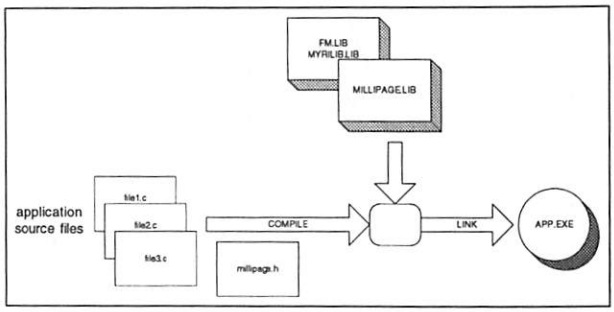


Figure 4: Preparing an application for execution on top of MILLIPAGE. The final executable includes both the application code and the MILLIPAGE libraries.

3.5 FastMessages

MILLIPAGE uses FM on Myrinet as its communication layer. FM was developed at the University of Illinois as a low latency messaging layer, working on fast networking media such as Myrinet. We measured a roundtrip delay of 25 usec for small messages (200 bytes) and 180 usec for 4 KB messages. FM achieves network bandwidth higher than 1 GB/sec on our switched Myrinet LAN.

FM provides a reliable and FIFO ordered messaging service. Its high performance is due to two main features. First, it does not switch between user mode and kernel mode, but rather transfers data directly to and from the user space. Second, it minimizes buffer copying of messages. When a send operation is initiated by the user process, FM verifies that there is sufficient space in the network buffers at the target network adapter. Then the message is read directly from the user space, through the local adapter to the target network adapter card. Using DMA, the message is copied at the receiver side from the buffers of the network adapter card to the FM reserved (and pinned) memory. The receiver should then poll in order to check that the message has arrived and can be processed by its handler.

Although we measured low latencies and high bandwidth for messages that were sent using FM, we confronted a problem caused by FM's polling policy. When coordination of the send-receive operations is possible, e.g., in message-passing interfaces such as MPI and PVM, the sender thread and the receiving

```
On Read or Write Fault
pmsg→event = myEvent;
pmsg→type = {READ or WRITE}_REQUEST;
pmsg→from = ME;
pmsg→addr = fault_addr;
Send pmsg to manager;
wait(myEvent);

Handle Read Request
if (access(pmsg→fault_addr) == ReadWrite)
    then set access(pmsg→fault_addr) to ReadOnly
pmsg→type = READ_REPLY;
Send pmsg to pmsg→from;
Send(pmsg→privbase,pmsg→pgsize) to pmsg→from;

Handle Write Request
Set access(pmsg→fault_addr) to NoAccess
pmsg→type = WRITE_REPLY;
Send pmsg to pmsg→from;
Send(pmsg→privbase,pmsg→pgsize) to pmsg→from;

Handle Read or Write Reply
Recv(pmsg→privbase,pmsg→pgsize);
Set access(pmsg→fault_addr) to ReadOnly/ReadWrite
SetEvent(pmsg→event);

Handle Invalidate Request
Set access(pmsg→fault_addr) to NoAccess
pmsg→type = INVALIDATE_REPLY;
Send pmsg to manager;

Manager: Translate(pmsg)
pmsg→base = get_minipage_base(pmsg→fault_addr);
pmsg→pgsize = get_minipage_size(pmsg→fault_addr);
pmsg→privbase = addr2priv(pmsg→base);

Manager: Handle Read Request
Translate(pmsg);
p = find_replica(pmsg→fault_addr);
Forward pmsg to p;

Manager: Handle Write Request
Translate(pmsg);
forall p in replicas of pmsg→fault_addr {
    pmsg→type = INVALIDATE_REQUEST;
    Send pmsg to p;
}
Manager: Handle Invalidate Reply
if got less than (#replicas - 1) replies then return;
pmsg→type = WRITE_REQUEST;
p = find_replica(pmsg→fault_addr);
Forward pmsg to p;
```

Figure 3: The complete protocol in MILLIPAGE. Note the simplicity of the DSM layer: no buffer copying, queuing, table lookup, or translation of any kind are required, except at the manager.

thread can be co-scheduled to achieve good timing of the polling action. Unfortunately, such coordination is impossible in DSM systems, since (mini)page faults occur in an unpredictable manner. When a thread faults and sends a (mini)page request to another process, this process will commonly be busy in application-related computation. In this situation, frequent polling will slow down the computation, whereas infrequent polling will cause large delays in receiving and handling the request. Since both responsiveness and efficiency have a major impact on DSM performance, we had to address this problem in our system design. We proceed below to describe our solutions.

3.5.1 Communication and DSM Server Threads

Application threads run as native kernel threads of the operating system. When started, they invoke a wrapper routine that installs the MILLIPAGE exception handler and calls the original main thread routine. Aside from this, the application code is executed as is and application threads experience no interference unless they fault.

In addition to the application threads, MILLIPAGE spawns three threads that are in charge of maintaining the memory consistency and servicing requests: two DSM server threads and a millisecond timer thread. One DSM server thread, called the *poller*, is constantly polling for messages in a busy loop. It promptly serves received messages, then continues to poll. The *poller* runs at a low priority; it does not consume cpu cycles when required by application threads. Another thread, called the *sweeper*, differs from the *poller* only in that it waits on an event before it issues a single poll.

Ideally, we would let the *sweeper* sleep for periods of a few hundred microseconds, maximizing responsiveness without overloading the cpu. However, high resolution timers are provided in Windows-NT only through *multimedia timers*, of which the finest resolution is 1ms. Therefore, a third thread, the *timer*, was used to set up an event to wake up the *sweeper* once in every millisecond. For higher accuracy, and since it consumes, over a period of a millisecond, very little time, the priority of the *sweeper* was set to exceed that of the application threads. Note that the multimedia *timer* thread runs in high priority as well.

Our measurements showed that the deviation in the time between timer events is extreme: most of them appear either within several tens of microseconds, which overloads the cpu, or take several mil-

liseconds, which degrades responsiveness. We have found that this anomaly of the Windows-NT timers has been recently reported in [11], where a standard deviation of 955 microseconds was measured for 1ms timers on similar hardware - nearly equal to the mean! Since polling is in the core of our system, the timer inaccuracy significantly affected the responsiveness of the DSM server threads, and thus the overall performance of MILLIPAGE.

4 Performance Evaluation

In this section we discuss the performance implications of using MULTIVIEW and the performance of the MILLIPAGE system. Our testbed environment consists of a network of eight Pentium II 300Mhz uniprocessor machines, running Windows-NT Workstation 4.0 SP3. Each machine has 128 Mbytes of RAM. The architecture page size is 4 KB. The cluster is interconnected by a switched Myrinet LAN [4]. The Myrinet drivers were taken from the HPVM release 1.0 for NT [8].

4.1 MultiView Limitations

In order to measure the overhead of using MULTIVIEW we used a standalone test application which is not related to the MILLIPAGE system. In this way the overhead of MULTIVIEW could be distinguished from that of other implementation-related sources.

Our test application allocates an array of characters (bytes). The array resides in minipages of equal size. The number of minipages in each page is equal to the number of views. The main application routine iteratively traverses the array, reading each element (from first to last) exactly once in each iteration. We experiment with two parameters: the size of the array (which represents the size of the shared memory), N, and the number of views n.

As expected, the total size of committed memory increases with the size of the allocated region, independent of the number of views. We were limited, however, by the size of the available virtual address space, which stands at about 1.63 GB. Thus, we could only experiment with $n \leq 1.63\text{Gig}/N$ (e.g., for $N = 16\text{MB}$ we could set up to $n = 104$ views).

For $N = 512\text{KB}$ we hardly noticed any overhead. For $1 \leq n \leq 32$ the measured overhead is always less than 4% for $512\text{KB} \leq N \leq 16\text{MB}$. Note that $n = 32$ means a sharing granularity of 128 bytes.

In order to study the limitations of the MULTIVIEW technique we also experimented with a very large number of views, up to 1664. The results show

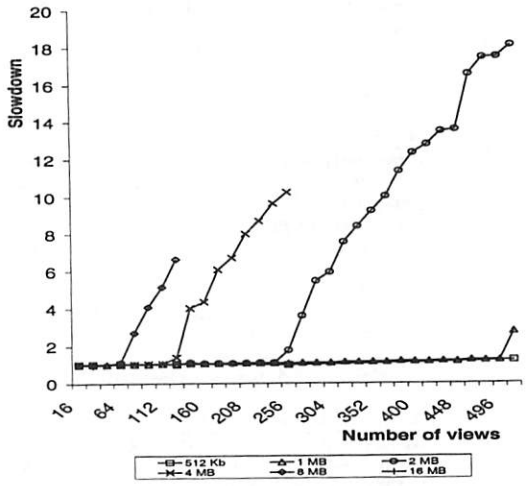


Figure 5: Overheads of MULTIVIEW. Note that the breaking-points where the overheads become substantial depend on the size of the shared memory in an inversely linear fashion.

that, at certain *breaking-points*, the overhead of using many views becomes substantial and may lead to severe slowdown. Figure 5 summarizes the results up to 512 views (minipages of size 8 bytes). Taking a closer look at the graphs, we could make several observations. First, for each N, beyond the breaking-point, the overhead increases linearly with the number of views. Second, beyond their respective breaking-points, the graphs for all N increase with the same slope. Third, the breaking-points themselves depend on N in an approximately inversely-linear fashion: they appear at the points where $n \cdot N = 512$ (N in MB). Finally, when we tried to allocate a large N and use only a fraction of it, the breaking-point appeared earlier than in the case where only the accessed fraction was allocated.

Our first explanation is that the slowdown is related to the enormous increase in TLB misses, and to the size of the cache. The number of active PT entries (not MPT) at the breaking points becomes 128K. The TLB size in the Pentium II is 64 data entries and 32 code entries. A PTE is four bytes in size, and the PTEs are cachable. A TLB miss and a 1st level cache miss cause a single stall; an increase there may thus explain the 1–4% slowdown we experienced with smaller N and n, but cannot possibly explain the slowdown that appears beyond the breaking-points.

The size of the 2nd level cache in our machines is 512KB, thus the breaking-points occur precisely when the PTEs can no longer be cached there. The cache is physically tagged. Attempting to access a new minipage causes the virtual memory translation mechanism to search for a new PTE, which, when the cache is congested, causes a miss. In the extreme situations that we tested, beyond the breaking-points, the cache misses caused by the missing PTEs dominate the cache activity.

We note here that other factors might also have affected the performance of MULTIVIEW. One example is a possible performance degradation caused by overloading the operating system's internal data structures, keeping track of extensive data mapping. Another example is the quicker cache exhaustion that may occur in virtually tagged caches.

As we describe later in this section, the applications in our benchmark suite all use less than 32 views and thus the overhead they experience is negligible.

4.2 Basic Costs in MILLIPAGE

A major goal in MILLIPAGE's design was to minimize the DSM protocol time by implementing a thin protocol layer. Table 1 shows the measured times (costs) of some basic operations that are part of the DSM protocols in MILLIPAGE.

operation	μs
access fault	26
get protection	7
set protection	12
header message send/recv (32 bytes)	12
a data message send/recv (0.5 KB)	22
a data message send/recv (1 KB)	34
a data message send/recv (4 KB)	90
minipage translation (MPT lookup)	7

Table 1: Cost of basic operations in MILLIPAGE.

The time it takes to bring in a page for reading in MILLIPAGE is 204 μs for minipages of size 128 byte, and 314 μs for minipages of size 4 KB. The difference in arrival times for a minipage request arriving in a single hop as opposed to two hops was slight. The time it takes to bring in a page for writing in MILLIPAGE is 212–366 μs for 128 bytes minipages, and 327–480 μs for 4KB minipages. These times vary according to the number of read copies that should be invalidated prior to serving the write request.

A *barrier* between 1 to 8 hosts takes 59–153 μs (linearly in the number of hosts) and a *lock* followed by an *unlock* operation takes 67–80 usec.

In addition, we measured diff creation time in our setting. Our measurements show that a run-length diff operation (as described in [5]) for 4KB page takes 250 μs and decreases linearly with the size of the page. Obviously, this time is not negligible, and would have dominated the overhead if it were required in the DSM protocol.

4.3 Applications

This section presents the results of parallel execution of five benchmark applications on the MILLIPAGE system. Our application suite consists of: Waternsquad (WATER) and LU-contiguous (LU) from - SPLASH-2 [25]; Integer-Sort (IS) from the NAS parallel benchmarks [2]; Successive Over Relaxation - (SOR) and the Traveling Salesperson Problem (TSP) from the Treadmarks [13] benchmark applications.

Table 2 summarizes application information such as data sets, shared memory size, and the sharing granularity. As can be seen, different applications naturally use minipages of different sizes, which in turn dictates the number of views as explained earlier in Section 2.

The code for memory allocation in three of the applications was slightly modified in order to equate the allocations and the sharing units.

In the original code for WATER, all the molecules are stored in a single array (VAR) and are referenced via pointers. We altered the main function so that each molecule will be allocated separately.

IS allocates a shared portion of memory where the keys reside. The array is relatively small and is divided into regions of equal size where each host is in charge of another region. We modified the allocation routine to have these regions allocated separately and thus reside in different minipages.

TSP allocates a global memory structure that contains an array of tours. Each tour (`TourElement`) is of size 148 bytes and each tour is manipulated exclusively by one of the tasks. We extracted the array out of the global memory structure, leaving there only a pointer. We then allocated each tour independently so that each one resides in a separate minipage.

There was no need to modify SOR, as it uses a matrix which is allocated row by row. The granularity of a row is suitable as the sharing unit, so the size of a row may determine that of a minipage. Similarly, it was not necessary to modify LU, as it

	Input Set	Shared Mem. Size	Num. views	Sharing Granularity	Barr.	Locks
SOR	32768x64 matrices	8 MB	16	a row, 256 bytes	21	-
IS	2^{23} numbers, 2^9 values	2 KB	8	256 bytes	90	-
WATER	512 molecules	336 KB	6	a molecule, 672 bytes	29	6720
LU	1024x1024 mat., 32x32 blocks	8 MB	1	a block, 4 KB	577	-
TSP	19 cities, recursion level 12	785KB	27	a tour, 148 bytes	3	681

Table 2: Application Suite.

builds a matrix by allocating sub-blocks, each of size $32 \times 32 \times |int| = 4$KB. Since the granularity of these sub-blocks is suitable as the sharing unit, the size of a minipage may be set equal to that of a 4KB page.

4.3.1 Speedups

Figure 6 summarizes the speedups on MILLIPAGE for each of the applications, when executing on 1 to 8 processors.

When examining the results, one should keep in mind that because of the FM polling problem and the large-grain timers described in Section 3.5, our system suffers from relatively high delays in servicing minipage requests. Since the resolution of the operating system timers is constrained to 1 ms (and is extremely inaccurate, see Section 3.5.1 and [11]), we experienced an average delay of about 750 μs for minipage requests. Only about a third of the delay comes from the DSM layer (see Section 4.2 above), while the rest is due to the slow response of the server thread: an average of more than 500 μs.

Despite the above, our initial experience with MILLIPAGE shows encouraging results. IS and SOR achieved speedups that are close to linear: the efficient resolution of false sharing led to a relatively small communication volume. WATER's performance was comparable to that achieved in reduced consistency systems. This performance was achieved by *chunking* molecules in larger minipages, a method that we describe later in this section.

LU achieved relatively good results, mainly due to the thin DSM layer which reduces the protocol overhead. In addition, in order to minimize the large minipage service delays explained above, we inserted two prefetch calls during the LU computation. We believe that this prefetch mechanism will not be needed once the FM polling problem is resolved, and/or the operating system timer resolution is refined.

False sharing was resolved in TSP, except for a single data race for updating the minimal tour found so far. Although the modification of this variable is protected by means of mutual exclusion, it is frequently read through an unprotected section. We changed a single code line in which this variable is updated, so that it pushes readable copies of the new value to all hosts. It is instructive to consider the minipage allocation size, which is equal to that of a tour element: 148 bytes. Providing granularity of 128 or 256 bytes ("cleaner" numbers that divide a page size) may involve a large increase in false sharing due to the pattern in which TSP assigns tours to processors: two adjacent tours are often assigned to two different processors.

4.4 Chunking

In the tradeoff between false sharing and aggregation we found that it is sometimes better to use granularity larger than the sharing unit size, with the cost of some additional false sharing. The main reason is the longer time it takes to bring in a relatively large portion of the memory when fine granularity is employed. Aggregation may reduce the number of DSM protocol calls, number of messages, page protection changes, and other sources of overhead, thus improving performance.

In our test applications we found WATER to behave much worse when we allocated each molecule in a separate minipage, than when several of them were chunked into a larger minipage. At the beginning of each iteration of WATER, each processor brings in the entire molecules' structure, namely, the *read phase*. When the allocation is done in granularity of molecules, the read phase takes a long time to complete due to the large number of minipage-faults that should be served. Despite the fact that false sharing is avoided for the computation which follows the read phase, this phase has a major impact on degrading the speedup. We therefore set a switch in MILLIPAGE, called *chunking level*, that makes MILLI-

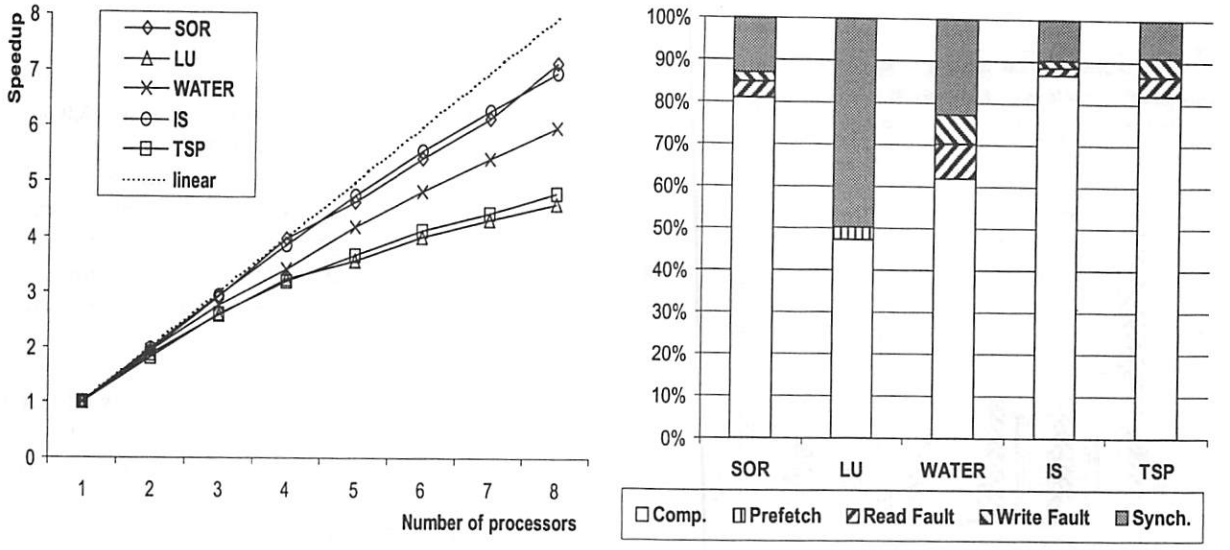


Figure 6: Summary of performance results. The breakdown graph on the right (for eight hosts) proves that the efficient resolution of false sharing results in a low communication volume, which shows itself in small total service times for faults. The total service time will further decrease once the polling and timer resolution problems are solved (see Section 3.5).

PAGE aggregate allocations in larger minipages.

Increasing the chunking level causes our manager to report more *competing requests*, i.e., requests for a certain minipage that are enqueued while a previous request is being served. When there was no chunking, 21 competing requests were reported. This surprised us at first, as false-sharing had been completely eliminated, and we expected no competing requests whatsoever. However, [17] already reported that there is a Write-Read data-race in WATER. Apparently this data-race is the cause for the competing requests reported by our manager.

We experimented with chunking in WATER for the shared molecules structure, setting the chunking level to increase from 1 to 6. We also ran WATER with no false-sharing control, so molecules were allocated in the traditional way; i.e., in minipages of a page size, disregarding minipage boundaries. As expected, the number of competing requests increases with the chunking level, reaching up to 601 when no false-sharing control is employed. From Figure 7 we conclude that the best performance is achieved for a chunking level of 4 or 5.

It is interesting to compare our findings with those of Shasta, as reported in [19]. They also found WATER to benefit from a relatively coarse granularity. However, they set the granularity level to 2048 bytes, while we found the optimum in a larger granularity level, 2688 or 3360 bytes. We expect that when the FM polling problem is solved (see Section 3.5), the optimal chunking level will decrease, in which case we may find it closer to that found by Shasta.

5 Discussion and Future Work

Prior to this work, the notion of a page-based DSMs which use page protection mechanisms was perceived as contradicting that of sharing data in fine granularity, and tightly coupled with that of false-sharing. Although relaxed consistency memories are successful in solving the problem, they require complicated protocols which introduce new sources of overhead.

In this paper we proposed a new method, called MULTIVIEW, that unbinds the (seemingly) tight connection between page-based DSMs and the need to share data in granularity of pages. We describe one realization of MULTIVIEW in a DSM system called MILLIPAGE. In addition to the MULTIVIEW implementation, the design of MILLIPAGE gives rise to other important issues such as the notion of a thin-layer DSM and the integration of fast messaging layers.

The MULTIVIEW technique and its implementation in MILLIPAGE open a new avenue of research directions. Work in these directions is already under way. We proceed in this section to mention only a few of them.

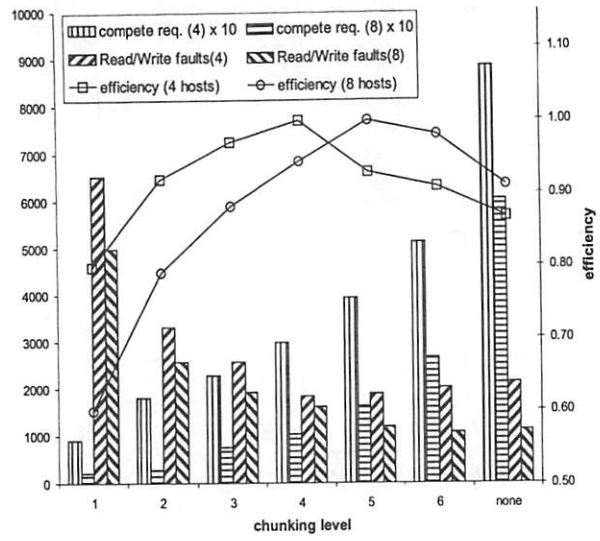


Figure 7: The effect of chunking in WATER. The optimal speedup is obtained at the chunking level of 4 for four hosts and 5 for eight hosts. Note the opposite tendencies of false-sharing and remote requests, represented by the number of competing requests and the number of Read/Write faults, respectively. The efficiency, which is given relative to the best chunking level, is determined by this tradeoff.

Reduced-Consistency Protocols

When the minipages defined for a certain application are larger than the sharing unit, i.e., the chunking level is set higher than one (see Section 4.4), performance may benefit from employing reduced-consistency protocols such as Lazy Release Consistency [12, 26]. Thus, chunking reduces the overhead involved in fine-grain operation, while false-sharing is eliminated through the reduced consistency protocol. The overhead involved in the reduced consistency protocol itself is small compared to that measured in traditional page-based systems, due to the smaller page size.

Compiler Work

While we chose to use only the operating system API, we believe that much can still be done through the compiler. A compiler can map and re-map variables to views, optimize accesses through those views, minimize PT usage, reduce TLB misses, etc. Large static variables can be distributed carefully among views in order to tune the granularity level with the access pattern.

Composed-Views

Complex data structures (such as multi-dimensional arrays) may be stored in groups of minipages. It might be helpful for an application to access these structures using different views at different stages. Higher level views may be associated with groups of lower level views, or groups of minipages. Obviously, the access permissions to such a *composed-view* should be set to the least of the access permissions of its components.

One good example where composed-views (or compiler work) can be used is the chunking-level in WATER, as discussed in Section 4.4. Clearly, the read phase in WATER could benefit from a coarse grain operation mode, whereas the later write phase would accelerate in a fine grain mode due to the elimination of false-sharing. Hence, replacing the current compromise by arbitration between fine-grained and coarser-grained views would speed up the computation.

Access Locality in the Page Table

The main weakness of MULTIVIEW is the lost of locality in using PTEs due to the unusual memory layout (see Section 4.1). Nevertheless, locality is not completely lost, but is preserved across views. A future work for integrating minipages inside the operating system may consider altering the PT data structure to exploit this memory layout.

Global Memory Systems

Recently, there has been a lot of research in utilizing remote memories for storing memory pages, thus establishing an additional stage in the memory hierarchy. It has been shown that using subpages as the transfer units leads to substantial performance boost [10]. We believe that MULTIVIEW can be used for implementing subpages in global memory systems.

Acknowledgments

We thank the referees, whose comments helped us improve the paper. We are grateful to Roy Friedman for his valuable advises throughout this work. We thank Evan Speight for providing us with benchmark applications from Brazos. We cheer the HPVM team from Illinois for developing and releasing the stable Myrinet drivers. Thanks to their drivers, we have gotten our Myrinet LAN working in just a short time.

References

[1] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A comparison of entry consistency and lazy release consistency implementations. In *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, pages 26–37, February 1996.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report RNR-91-002, NASA Ames, August 1991.

[3] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, February 1993.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[5] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, October 1991.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Trans. on Computer Systems*, 13(3):205–243, August 1995.

[7] S. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proc. of the 20th Annual Int'l Symp. on Computer Architecture (ISCA'93)*, pages 144–155, May 1993.

[8] HPVM 1.0 for Windows NT 4.0 on x86. CSAG, University of Illinois. http://www-csag.cs.uiuc.edu/projects/hpvm/sw-distributions/.

[9] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, June 1996.

[10] H. A. Jamrozik, M. J. Feeley, G. M. Voelker, J. Evans II, A. R. Karlin, H. M. Levy, and M. K. Vernon. Reduing Network Latency Using Subpages in a Global Memory Environment. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 258–267, October 1996.

[11] M. B. Jones and J. Regehr. Issues in Using Commodity Operating Systems for Time-Dependent Tasks: Experiences from a Study of Windows NT. In *Proceedings of the Eighth International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'98)*, pages 107–110, July 1998.

[12] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[13] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[14] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proc. of the 1988 Int'l Conf. on Parallel Processing (ICPP'88)*, volume II, pages 94–101, August 1988.

[15] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proc. of the 5th Annual ACM Symp. on Principles of Distributed Computing (PODC'86)*, pages 229–239, August 1986.

[16] S. Pakin, V. Karamacheti, and A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 5(2):60–73, 1997.

[17] D. Perkovic and P. Keleher. Online data-race detection via coherency guarantees. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pages 47–57, October 1996.

[18] D. J. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, October 1997.

[19] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 174–185, October 1996.

[20] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-Grain Access Control for Distributed Shared Memory. In *Proc. of the 6th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVI)*, pages 297–306, October 1994.

[21] David A. Solomon. *Inside Windows-NT, 2nd Edition.* Microsoft Press, 1998.

[22] E. Speight and J.K. Bennett. Brazos: A Third Generation DSM System. In *First Usenix-NT Workshop*, pages 95–106, August 1997.

[23] M. Swanson, L. Stroller, and J. B. Carter. Making distributed shared memory simple, yet efficient. In *Proc. of the 3rd Int'l Workshop on High-Level*

Parallel Programming Models and Supportive Environments, pages 2–13, March 1998.

[24] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, May 1992.

[25] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture (ISCA'95)*, June 1995.

[26] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, October 1996.

[27] Y. Zhou, L. Iftode, K. Li, J. P. Singh, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed consistency and coherence granularity in dsm systems: A performance evaluation. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 193–205, June 1997.

Optimizing the Idle Task and Other MMU Tricks

Cort Dougan Paul Mackerras Victor Yodaiken

Department of Computer Science
New Mexico Institute of Technology
Socorro, NM 87801
{cort,yodaiken}@cs.nmt.edu *
paulus@cs.anu.edu.au

Abstract

In highly cached and pipelined machines, operating system performance, and aggregate user/system performance, is enormously sensitive to small changes in cache and TLB hit rates. We have implemented a variety of changes in the memory management of a native port of the Linux operating system to the PowerPC architecture in an effort to improve performance. Our results show that careful design to minimize the OS caching footprint, to shorten critical code paths in page fault handling, and to otherwise take full advantage of the memory management hardware can have dramatic effects on performance. Our results also show that the operating system can intelligently manage MMU resources as well or better than hardware can and suggest that complex hardware MMU assistance may not be the most appropriate use of scarce chip area. Comparative benchmarks show that our optimizations result in kernel performance that is significantly better than other monolithic kernels for the same architecture and highlight the distance that micro-kernel designs will have to travel to approach the performance of a reasonably efficient monolithic kernel.

1 Motivation

In the development of the PowerPC port of the Linux operating system, we have carried out a series of optimizations that has improved application wall-clock performance by anywhere from 10% to several orders of magnitude. According to the LmBench [5] benchmark, Linux/PPC is now twice as fast as IBM's AIX/PPC operating system and between 10 and 120 times faster than Apple's Mach based MkLinux/PPC and Rhapsody/PPC operating systems. We have achieved this level of performance by extensive use of quantitative measures and detailed analysis of low level system performance — particularly regarding memory management. While many of our optimizations have been machine specific, most of our results can be easily transported to other modern architectures and, we believe, are interesting both to operating system developers and hardware designers.

Our optimization effort was constrained by a requirement that we retain compatibility with the main Linux kernel development effort. Thus, we did not consider optimizations that would have required major changes to the (in theory) machine independent Linux core. Given this constraint, memory management was the obvious starting point for our investigations, as the critical role of memory and cache behavior for modern processor designs is well known. For commodity PC systems, the slow main memory systems and buses intensify this effect. What we found was that system performance was enormously sensitive to apparently small changes in the organization of page tables, in how we control the translation look aside buffer (TLB) and apparently innocuous OS operations that weakened locality of memory references. We also found that having a repeatable set of benchmarks was an invaluable aid in overcoming intuitions about the critical performance issues.

Our main benchmarking tools were the LmBench program developed by Larry McVoy and the standard Linux benchmark: timing and instrumenting a complete recompile of the kernel. These benchmarks tested aspects of system behavior that experience has shown to be broadly indicative for a wide range of applications. That is, performance improvements on the benchmarks seem to correlate to wall-clock performance improvements in application code. Our benchmarks do, however, ignore some important system behaviors and we discuss this problem below.

The experiments we cover here are the following:

- Reducing the frequency of TLB and secondary page map buffer misses.
  - Reducing the OS TLB footprint.
  - Increasing efficiency of hashed page tables.
- Reducing the cost of TLB misses.
  - Fast TLB reload code.
  - Removing hash tables — the ultimate optimization.
- Reducing TLB and page map flush costs.

[1]This work was partially funded by Sandia National Labs Grant 29P-9-TG100

- Lazy purging of invalid entries.
- Trading off purge costs against increased misses.

- Optimizing the idle task!

2 Related Work

There has been surprisingly little published experimental work on OS memory management design. Two works that had a great deal of influence on our work were a series of papers by Bershad [7] advocating the use of *"superpages"* to reduce TLB contention and a paper by Liedtke [3] on performance issues in the development of the L4 microkernel. Our initial belief was that TLB contention would be a critical area for optimization and that the monolithic nature of the Linux kernel would allow us to gain much more than was possible for L4.

It is the current trend in chip design to keep TLB size small, especially compared to the size of working sets in modern applications. There are many proposed solutions to the problem of limited TLB reach, caused by the disparity between application access patterns and TLB size, but most of them require the addition and use of special purpose hardware. Even the simpler proposed solutions require that the hardware implement superpages.

Swanson proposes a mechanism in [10] that adds another level of indirection to page translation to create noncontiguous and unaligned superpages. This scheme makes superpage use far more convenient since the memory management system does not have to be designed around it (finding large, continuous, aligned areas of unused memory is not easy).

Since existing hardware designs are set and the trend in emerging designs is towards relatively small TLBs we cannot rely on superpage type solutions or larger TLBs. More effective ways of using the TLB and greater understanding of TLB access patterns must be found. Greater understanding of access patterns and better ways of using TLB would only augment systems with superpages or larger TLBs.

3 A Quick Introduction to the PPC Memory Management System

Our OS study covers the 32-bit PowerPC 603 [6] and 604 processors. Both machines provide a modified inverted page table memory management architecture. The standard translation between logical and physical addresses takes place as follows:

Program memory references are 32-bit *logical* addresses. The 4 high order bits of the logical address index a set of segment registers, each of which contains a 24-bit *"virtual segment identifier"*(VSID). The logical address is concatenated with the VSID to produce a *virtual* address. There is a translation look-aside buffer of cached virtual → physical translations and *hashed page tables* indexed by a (hashed) virtual address. The tables are organized into "buckets", each consisting of eight page table entries (PTEs). Each

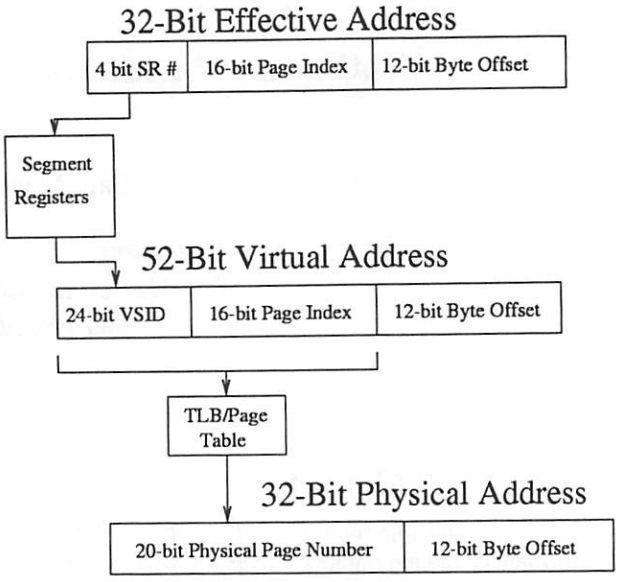


Figure 1: PowerPC hash table translation

PTE contains a 20-bit physical page address, a 24-bit virtual segment identifier (VSID) and permission and other housekeeping information. Once a TLB miss occurs, a hash function is computed on the virtual address to obtain the index of a bucket. If no matching entry is found in this bucket, a secondary hash function is computed to find the index of an overflow bucket. If no entry is found in either bucket, the OS must determine further action. On the 604, a TLB miss causes the hardware to compute the hash function and search the hash table. If no match is found, a page fault interrupt is generated and a software handler is started. On the 603, there are registers to assist hashing even though the hardware does not require software to store PTEs in a hash table. Since a TLB miss is handled in hardware, the 604 has a hash-table miss interrupt rather than a TLB miss interrupt.

The PowerPC also offers an alternative translation from logical to physical that bypasses the TLB/hash-table paging mechanism. When a logical address is referenced, the processor begins the page lookup and, in parallel, begins an operation called *block address translation*(BAT). Block address translation depends on eight BAT registers: four data and four instruction. The BAT registers associate virtual blocks of 128K or more with physical segments. If a translation via the BAT registers succeeds, the page table translation is abandoned.

4 Performance Measurement

Benchmarks and tests of performance were made on a number of PowerPC processors and machine types (PReP and PowerMac) to reduce the amount any specific machine would affect measurements. We used 32M of RAM in each machine tested. This way, the ratio of RAM size to PTEs in the hash table to TLB entries remained the same. Each of the test results comes from more than 10 of the benchmark

runs averaged. We ignore benchmark differences that were sporadic even though we believe this understates the extent of our optimizations.

Tests were made using LmBench [5]. We also used the informal Linux benchmark of compiling the kernel, which is a traditional measure of Linux performance. The mix of process creation, file I/O, and computation in the kernel compile is a good guess at a typical user load in a system used for program development.

Performance comparisons were made against various versions of the kernel. In our evaluations we compare the kernel against the original version without the optimizations discussed in this paper. This highlights each optimizations alone without the others. This lets us look more closely at how each change affects the kernel by itself before comparing all optimizations in aggregate. This turned out to be very useful as many optimizations did not interact as we expected them to and the end effect was not the sum off all the optimizations. Some optimizations even cancelled the effect of previous ones. So, measurements are relative to the original (unoptimized) kernel versus only the specific optimization being discussed for comparison unless otherwise noted.

Finally, we gathered low-level statistics with the PPC 604 hardware monitor. Using this monitor we were able to characterize the system's behavior in great detail by counting every TLB and cache miss, whether data or instruction. Software counters on the 603 were used to serve in much the same fashion as hardware performance monitoring hardware on the 604, but with a less fine-grained scope.

We make many references to the 603 software versus the 604 hardware TLB reload mechanism. In this context, when we refer to the 604 we mean the 604 style of TLB reloads (in hardware) which includes the 750 and 601.

5 Reducing the Frequency of TLB Misses

The 603 generates software handled interrupts on a TLB miss. It takes 32 cycles simply to invoke and return from the handler — ignoring the costs of actually reloading the TLB. The 604 PPC executes a hardware hash table search on a TLB miss. If the PTE is in the hash table (the PowerPC page-table), the cost of the hardware reload can take up to 120 instruction cycles (measured experimentally) and 16 memory accesses. If the hash table does not contain the PTE, the 604 generates a software handled interrupt that adds at least 91 more cycles to just invoke the handler. With interrupt overhead this high, TLB reloads will be expensive no matter how much we optimize the TLB reload code itself. With this motivation, we thought it worthwhile to reduce the frequency of TLB misses as much as possible.

5.1 Reducing the OS TLB footprint

Bershad [7] and others have argued on theoretical grounds that "superpages" and other mechanisms for reducing the OS TLB footprint can greatly improve performance. Indeed, we found that 33% of the TLB entries under Linux/PPC were for kernel text, data and I/O pages. The PowerPC 603 TLB has 128 entries and the 604 has 256 entries, so allocating a third of the entries to the OS should have a significant effect on performance. While some processor architectures (MIPS [2]) directly support superpage schemes, the PPC does not. There are too few BAT registers and their granularity is too coarse for a straightforward superpage implementation. The BATs can, however, still be used to reduce the number of entries taken up in the page-table and, therefore, reduce the number of TLB misses. Since user processes are often ephemeral and large block sizes for each user process would waste physical memory, we decided to begin by using the BAT mapping only for kernel address space.

Linux, like many other UNIX variants, divides each user processes virtual address space into two fixed regions: one for user code and data and one for the kernel. On a 32 bit machine, the Linux kernel usually resides at virtual address `0xc0000000` and virtual addresses from `0xc0000000` to `0xffffffff` are reserved for kernel text/data and I/O space. We began by mapping all of kernel memory with PTEs. We quickly decided we could reduce the overhead of the OS by mapping the kernel text and data with the BATs. The kernel mappings for these addresses do not change and the kernel code and static data occupy a single contiguous chunk of physical memory. So, a single BAT entry maps this entire address space. Note that one side effect of mapping kernel space via the BATs is that the hash tables and backing page tables do not take any TLB space. Mapping the hash table and page-tables is given to us for free so we don't have to worry about recursively faulting on a TLB miss.

Using the BAT registers to map kernel space on the kernel compile we measure a 10% reduction in TLB misses (from 219 million to 197 million TLB misses on average) and a 20% reduction in hash table misses (from an average 1 million hash table misses to 813 thousand hash table misses) during our benchmarks. The percentage of TLB slots occupied by the kernel dropped to near zero — the high water mark we have measured for kernel PTE use is four entries. The kernel compile benchmark showed a 20% reduction in wall-clock time - from 10 to 8 minutes. Using the BAT registers to map the I/O space did not improve these measures significantly. The applications we examined rarely accessed a large number of I/O addresses in a short time so it is rare that the TLB entries are mapping I/O areas since they are quickly displaced by other mappings. We have considered having the kernel dedicate a BAT mapping to the frame buffer itself so programs such as X do not compete constantly with other applications or the kernel for TLB space. In fact, the entire mechanism could be done per-process with a call to `ioremap()` and giving each process its own data BAT entry that could be switched during a context switch.

Much to our chagrin, nearly all the measured performance improvements we found from using the BAT registers evaporated when TLB miss handling was optimized. That is, the TLB misses caused by kernel - user contention

are few enough so that optimizing reloads makes the cost of handling these reloads minimal — for the benchmarks we tried. In light of Talluri [11], however, it's quite possible that our benchmarks do not represent applications that really stress TLB capacity. More aggressive use of the TLB, such as several applications using many TLB entries running concurrently would possibly show an even greater performance gain. Not coincidentally, this optimizes for the situation of several processes running in separate memory contexts (not threads) which is the typical load on a multiuser system.

5.2 Increasing the Efficiency of Hashed Page Tables

The core of Linux memory management is based on the $x86$ two-level page tables. We could change the organization of the PTEs in these tables to match the requirements of the PPC architecture (a hash table instead of a two-level page table), but we were committed to using these page tables as the initial source of PTE's due to the design of Linux. Note that any PPC OS must have a data structure to serve this function, because the PTEs that do not fit in either the primary or overflow bucket must be stored somewhere. It is possible, but impractical, to resize the hash table when both buckets overflow. Various techniques for handling overflow are discussed in [8] and [12]. A reasonable PPC OS must minimize the number of overflows in the hash table so the cost of handling overflows was not a serious concern for us. Instead, we focused on reducing the contention in the hash table to increase the efficiency of the hash table which reduces the number of overflows. Our original strategy for both the 603 and 604 processors was to use the hash table as a second level TLB cache and, thus, it became important to reduce hash table "hot spots" and overflow.

The obvious strategy is to derive VSIDs from the process identifier so that each process has a distinct virtual address space. Keep in mind that the hardware will treat each set of VSIDs as a separate address space. Multiplying the process id by a small non-power-of-two constant proved to be necessary in order to scatter PTEs within the hash table. Note that the logical address spaces of processes tend to be similar so the hash functions rely on the VSIDs to provide variation. We tuned the VSID generation algorithm by making Linux keep a hash table miss histogram and adjusting the constant until hot-spots disappeared. We began with 37% use of the hash table and were able to bring that up to an average of 57% with the hash table containing both user and kernel PTE's. After removing the kernel PTE's from the hash table we were eventually able to achieve 75% use of the hash table with find tuning of the constant.

6 Reducing the Cost of TLB and Hash Table Misses

6.1 Fast Reload Code

On an interrupt, the PowerPC turns off memory management and invokes a handler using physical addresses. Originally, we turned the MMU on, saved state and jumped to fault handlers written in C to search the hash table for the appropriate PTE. To speed the TLB reload we rewrote these handlers in assembly and hand optimized the TLB and hash table miss exception code for both 603 and 604 processors. The new handlers ran with the memory management hardware off and we tried to make sure that the reload code path was as short as possible.

Careful coding of miss handlers proved to be worth the effort. On an interrupt, the PPC turns off memory management and swaps 4 general purpose registers with 4 interrupt handling registers on a TLB miss. We rewrote the TLB miss code to use only these registers in the common case. Following the example of the Linux/SPARC developers, we also hand scheduled the code to minimize pipeline hiccups. The Linux PTE tree is sufficiently simple that searching for a PTE in the tree can be done conveniently with the MMU disabled, in assembly code, and taking three loads in the worst case. If the PTE cannot be found at all or if the page is not in memory, we turn on memory management, save additional context and jump to C code.

These changes produced a 33% reduction in context switch time and reduced communication latencies by 15% as measured with LmBench. User code showed an improvement of 15% in general when measured by wall-clock time.

6.2 Improving Hash Tables Away

The 603 databook recommends using hardware hashing assists to emulate the 604 behavior on the 603. Following this recommendation, the early Linux/PPC TLB miss handler code searched the hash table for a matching PTE. If no match was found, software would emulate a hash table miss interrupt and the code would execute as if it were on a 604 that had done a search in hardware. Our conjecture was that this approach simply added another level of indirection and would cause cache misses as the software stumbled about the hash table.

The optimization we tried was to eliminate any use of the hash table and to have the TLB miss handler go directly to the Linux PTE tree. By following this strategy we make a 180MHz 603 keep pace with a 185MHz 604 despite the two times larger L1 cache and TLB in the 604. In fact, on some LmBench points, the 180MHz 603 kept pace with a 200MHz 604 on a machine with significantly faster main memory and a better board design. Unfortunately, the 604 does not permit software to reload the TLB directly, which would allow us to make this optimization on the 604. The end result of these changes was a kernel compile time reduced by 5%.

processor	pstart	ctxsw	pipe lat.	pipe bw	file reread
603 180MHz (htab)	1.8s	4 μs	17 μs	69 MB/s	33 MB/s
603 180MHz (no htab)	1.7s	3 μs	19 μs	73 MB/s	36 MB/s
604 185MHz	1.6s	4 μs	21 μs	88 MB/s	39 MB/s
604 200MHz	1.6s	4 μs	20 μs	92 MB/s	41 MB/s

Table 1: LmBench summary for direct (bypassing hash table) TLB reloads

Using software TLB reloads which are available on many platforms, such as the Alpha [9], MIPS [2] and UltraSPARC, allows the operating system designer to consider many different page-table data structures (such as clustered page tables [11]). If the hardware doesn't constrain the choices many optimizations can be made depending on the type of system and typical load the system is under.

7 Reducing TLB and Hash Table Flush Costs

Because the 604 requires us to use the hash table, any TLB flush must be accompanied by a hash table flush. Linux flushes all or part of a process's entries in the TLB frequently, such as when mapping new addresses into a process, doing an `exec()` or `fork()` and when a dynamically linked Linux process is started, the process must remap its address space to incorporate shared libraries. In this context, a TLB flush is actually a TLB invalidate since we updated the page-table PTE dirty/modified bits when we loaded the PTE into the hash table. In the worst case, the search requires 16 memory references (2 hash table buckets, containing 8 PTE's each) for each PTE being flushed. It is not uncommon for ranges of 40 — 110 pages to be flushed in one shot.

The obvious strategy, and the first one we used, was for the OS to derive VSIDs from the process identifier (so that each process has a distinct virtual address space) and multiplying it by a scalar to scatter entries in the hash table. After doing this, we found that flushing the hash table was extremely expensive, we then came upon the idea of lazy TLB flushes. Our idea of how to do lazy TLB flushes was to keep a counter of memory-management contexts so we could provide unique numbers for use as VSID's instead of using the PID of a process. We reserved segments for the dynamically mapped parts of the kernel (static areas, data and text, are mapped by the BATs) and put a fixed VSID in these segments. When the kernel switched to a task its VSIDs could be loaded from the task structure into hardware registers by software.

When we needed to clear the TLB of all mappings associated with a particular task we only had to change the values (VSIDs) in the task structure and then update the hardware registers and increment the context counter. Even though there could be old PTEs from the previous context (previous VSIDs) in the hash table and TLB marked as valid PTEs (valid bit set in the PTE) their VSID's will not match any VSID's used by any process so incorrect matches won't be made. We could then keep a list of "zombie" VSID's (similar to "zombie" processes) that are marked as valid in the hash table but aren't actually used and clear them when hash table space became scarce.

What we finally settled on and implemented was different from what we had planned at first. Deciding when to really flush the old PTE's from the hash table and how to handle the "zombie list" was more complex than we wanted to deal with at first. Performance would also be inconsistent if we had to occasionally scan the hash table and invalidate "zombie" PTE's when we needed more space in the hash table. So, instead, we just allowed the hash table reload code to replace an entry when needed (not checking if it has a currently valid VSID or not). This gave a non-optimal replacement strategy in the hash table since we may replace real PTEs (have a currently active VSID) in the hash table even though there are PTEs that aren't being used (have the VSID of an old memory context).

We were later able to return to this idea of reducing the inefficiency of the hash table replacement algorithm (replacing unused PTE's from an abandoned memory management context marked as valid) by setting the idle task to reclaim zombie hash table entries by scanning the hash table when the cpu is idle and clearing the valid bit in zombie PTEs (physically invalidating those PTEs). This provided a nice balance of simplifying the actual low level assembly to reload the TLB and still maintaining a decent usage ratio of the hash table (zombie PTEs to in-use PTEs).

Without the code to reclaim zombie PTEs in the idle task, the ratio of hash table reloads to evicts (reloads that require a valid entry be replaced) was normally greater than 90%. Since the hot-spots were eliminated in the hash table, entries are scattered across all of the hash table and never invalidated. Invalidation in this case means the valid bit is never cleared even though the VSID may not match a current process. Very quickly the entire hash table fills up. Since the TLB reload code did not differentiate between the two types of invalid entries, it chose an arbitrary PTE to replace when reloading, replacing valid PTEs as well as zombie PTEs. With the reclaim code in the idle task, we saw a drastic decrease in the number of evicts. This is because the hash table reload code was usually able to find an empty TLB entry and was able to avoid replacing valid PTEs.

Our series of changes took hash table use up to 15% and finally down to around 5% since we effectively flush all the TLB entries of a process when doing large TLB flushes (by changing the VSID's of a process). Our optimization to re-

duce hot-spots in the hash table was not as significant since so few entries stayed in the hash table (about 600–700 out of 16384) at one time due to the flushing of ranges of PTEs. Even with this little of the hash table in use we measured 85% — 95% hit rates in the hash table on TLB misses. To increase the percentage of hash table use, we could have decreased the size of the hash table and free RAM for use by the system but in performing these benchmarks we decided to keep the hash table size fixed to make comparisons more meaningful. This choice makes the hash table look inefficient with some optimizations but the net gain in performance as measured by hit rate in the hash table and wall-clock time shows it is in fact an advantage.

Another advantage of the idle task invalidating PTEs was that TLB reload code was usually able to find an empty entry in the hash table during a reload. This reduced the number of evicts so the ratio of evicts to TLB reloads became 30% instead of the greater than 90% we were seeing before. This reduced number of evicts also left the hash table with more in-use PTEs so our usage of the hash table jumped to 1400–2200 from 600–700 entries, or to 15% from 5%. The hit rate in the hash table on a TLB miss also increased to as high as 98% from 85%.

Using lazy TLB flushes increased pipe throughput by 5 MB/s (from 71 MB/s) and reduced 8-process context switch times from 20μs to 17μs. However, the system continued to spend a great deal of time searching the hash table because for certain operations, the OS was attempting to clear a *range* of virtual addresses from the TLB and hash table. The OS must ensure that the new mappings are the only mappings for those virtual addresses in the TLB and hash table. The system call to change address space is `mmap` and LmBench showed `mmap()` latency to be more than 3 milliseconds. The kernel was clearing the range of addresses by searching the hash table for each PTE in turn. We fixed this problem by invalidating the whole memory management context of any process needing to invalidate more than a small set of pages. This effectively invalidates all of the TLB entries of this process. This was a cheap operation with the mechanism we used since it just involved a reset of the VSID whose amortized cost (later TLB reloads vs. cost of flushing specific PTEs) is much lower. Once the process received a new VSID and its old VSID was marked as zombie, all the process' PTEs in the TLB and hash table were automatically inactive. Of course, there is a performance penalty here as we invalidate some translations that could have remained valid, but using 20 pages as the cutoff point `mmap()` latency dropped to 41μ — an 80 times improvement. Pipe bandwidth increased noticeably and several latencies dropped as well. These changes come at no cost to the TLB hit rate since no more or fewer TLB misses occurred with the tunable parameter to flushing ranges of PTEs. This suggests that the TLB entries being invalidated along with target range of TLBs were not being used anyway - so there is no cost for losing them.

Table 2 shows the 603 doing software searches of the hash table and a 604 doing hardware searches with the effect of lazy TLB flushes. Note that the 603 hash table search is using software TLB miss handlers that emulate the 604 hardware search. This table shows the gain from avoiding expensive searches in the hash table when a simple resetting of the VSID's will do.

8 Cache Misuse on Page-Tables

Caching page-tables can be a misuse of the hardware. The typical case is that a program changes its working set, which adds a particular page. That page is referenced and its PTE is brought into the TLB from the page-tables and during that TLB reload the PTE is put in the data cache. On the 604 the PTE is also put into the hash table, which creates another cache entry. TLB reloads from page-tables are rare and caching those entries only pollutes the cache with addresses that won't be used for a long time.

Caching page tables makes sense only if we will make repeated references to adjacent entries, but this behavior does not occur nor is it common for page-table access patterns. For this to be common, we'd need to assume that either the page accesses (and PTEs used to complete those accesses) or cache accesses do not follow the principle of locality. The idea behind a TLB is that it's rare to change working sets and the same is true for a cache.

This is especially important under Linux/PPC since the hardware searched hash table (the PowerPC page-table) is actually used as a cache for the two level page table tree (similar to the SPARC and $x86$ tables). This makes it possible in the worst case for two separate tables to be searched in order to load one PTE into the TLB. This translates into 16 (hash table search and miss) + 2 (two level page table search) + 16 (finding a place to put the entry in the hash table) = 34 memory accesses in order to bring an entry into the hash table. If each one of these accesses is to a cached page there will be 18 new cache entries created (note that the first and second 16 hash table addresses accessed are the same so it is likely we will hit the cache on the second set of accesses).

By caching the page-tables we cause the TLB to pollute the cache with entries we're not likely to use soon. On the 604 it's possible to create two new cache entries that won't be used again due to accessing the hash table and the two-level page tables for the same PTE. This conflict is non-productive and causes many unnecessary cache misses. We have not yet performed experiments to quantify the number of cache misses caused by caching the page-tables but the results of our work so far suggests this has a dramatic impact on performance.

9 Idle Task Page Clearing

Clearing pages in the idle task is not a new concept but has not been popular due to its effect on the data cache. For the same reason we did not use the PowerPC instruction that clears entire cache lines at a time when we implemented *bzero*() and similar functions. This problem illustrates the effect the operating system has on perfor-

processor	mmap lat.	ctxsw	pipe lat.	pipe bw	file reread
603 133MHz	3240 μs	6 μs	34 μs	52 MB/s	26 MB/s
603 133MHz (lazy)	41 μs	6 μs	28 μs	57 MB/s	32 MB/s
604 185MHz	2733 μs	4 μs	22 μs	90 MB/s	38 MB/s
604 185MHz (tune)	33 μs	4 μs	21 μs	94 MB/s	41 MB/s

Table 2: LmBench summary for tunable TLB range flushing

mance due to its use of the cache. We began by clearing pages in the idle task without turning off the cache for those pages. These pages were then added to a list which `get_free_page()` then used to return pages that had already been cleared without having to waste time clearing the pages when they were requested. The kernel compile with this "optimization" took nearly twice as long to complete due to cache misses. Measurements with LmBench showed performance decreases as well. The performance loss from clearing pages was verified with hardware counters to be due to more cache misses.

We repeated the experiment by uncaching the pages before clearing them and not adding them to the list of cleared pages. This allowed us to see how much of a penalty clearing the pages incurred without having the effect of using those pages to speed `get_free_page()`. There was no performance loss or gain. This makes sense since the data cache was not affected because the pages being cleared were uncached and even after being cleared they weren't used to speed up `get_free_page()`. The number of cache misses didn't change from the kernel without the page clearing modifications. [2]

When the cache was turned off for pages being cleared and they were used in `get_free_page()` the system became much faster. This kept the cache from having useless entries put into it when `get_free_page()` had to clear the page itself when the code requesting the page never read those values (it shouldn't read them anyway). This suggests that it might be worthwhile to turn off the data and instruction cache during the idle task to avoid polluting the cache with any accesses done in the idle task. There's no need to worry about the idle task executing quickly, we're only concerned with switching out of it quickly when another task needs to run so caching isn't necessary.

We must always ensure that the overhead of an optimization doesn't outweigh any potential improvement in performance [11] [4]. In this case we did not incur great overhead when clearing pages. In fact, all data structures used to keep track of the cleared pages are lock free and interrupts are left enabled so there is no possibility of keeping control of the processor any longer than if we had not been clearing pages. Even when calling `get_free_page()` the only overhead is a check to see if there are any pre-cleared pages available. Our measurements with page-clearing on but not adding the pages to the list still costs us that check in `get_free_page()` so any potential overhead would have shown up. This is important since the idle task runs quite often even on a system heavily loaded with users compiling, editing, reading mail so a lot of I/O happens that must be waited for.

[2] On a SMP machine we might see conflicts due to accesses causing cache operations on other processors or, more likely, all these writes to memory using a great deal of the bus while the other processor needs it

10 Future Work

There is still more potential in the optimizations we've already made. We've made these changes on a step-by-step basis so we could evaluate each change and study not only how it changed performance but why. There are still many changes we've seen that we haven't actually studied in detail that we believe to be worthy of more study. These include better handling of the cache in certain parts of the operating system and cache preloads.

10.1 Locking the Cache

Our experiments show that not using the cache on certain data in critical sections of the operating system (particularly the idle task) can improve performance. One area worthy of more research would be locking the cache entirely in the idle task. Since all of the accesses in the idle task (instruction and data) aren't time critical there's no need to evict cache entries just to speed up the idle task.

10.2 Cache Preloads

Waiting on the cache to load values from memory is a big performance loss. Modern processors rely on the fact that the pipeline will be kept full as much as possible but mishandling the cache can easily prevent this. One easy way to overcome this problem is to provide hints to the hardware about access patterns. On the PowerPC and other architectures there are sets of instructions to do preloads of data cache entries before the actual access is done. This is a simple way, without large changes, to improve cache behavior and reduce stalls. We feel that we can make significant gains with intelligent use of cache preloads in context switching and interrupt entry code.

11 Analysis

The results presented show that it is possible to achieve and exceed the speed of hardware TLB reloads with software handlers. This speedup depends very much on how well

OS	Null syscall	ctx switch	pipe lat.	pipe bw
Linux/PPC	2 μs	6 μs	28 μs	52 MB/s
Unoptimized Linux/PPC	18 μs	28 μs	78 μs	36 MB/s
Rhapsody 5.0	15 μs	64 μs	161 μs	9 MB/s
MkLinux	19 μs	64 μs	235 μs	15 MB/s
AIX	11 μs	24 μs	89 μs	21 MB/s

All tests except AIX performed on a 133MHz 604 PowerMac 9500, AIX number from a 133MHz 604 IBM 43P.

Table 3: LmBench summary for Linux/PPC and other Operating Systems

tuned the reload code is and what data structures are used to store the PTEs. On the 603 we find it is not necessary to mirror the same hash table that the hardware assumes on the 604. We can actually speed things up by eliminating the hash table entirely.

By reducing the conflict between user and kernel space for TLB entries we're able to improve TLB hit rates and speed the system up in general by about 20%. Our experiments bypassing the cache during critical parts of the operating system where creating new cache entries would actually reduce performance shows great promise. Already we're seeing fewer cache misses by avoiding creating cache entries for the idle task and expect to see even fewer with changes to the TLB reload code to uncache the page tables. We've been able to avoid expensive TLB flushes through several optimizations that bring `mmap()` latency down to a reasonable value - an 80 times improvement by avoiding unnecessary searches through the hash table.

Our hash table hit rate on a TLB miss is 80% – 98% which demonstrates that the hash table is well managed to speed page translations. We cannot realistically expect any improvement over this hit rate so our 604 implementation of the MMU is near optimal. When compared with our optimizations of the 603 MMU using software searches we're able to get better performance on the 603 in some benchmarks than the 604 even though the 604 has double the size TLB and cache.

All these changes suggest that cache and TLB management is important and the OS designer must look deeper into the interaction of the access patterns of TLB and cache. It isn't wise to assume that caching a page will necessarily improve performance.

Though it has been claimed [1] that micro-kernel designs can be made to perform as well as monolithic designs our data (Table 3) suggests that monolithic designs need not remain a stationary target.

The work we've mentioned has brought Linux/PPC to excellent standing among commercial and non-commercial offerings for operating systems. Table 3 shows our system compares very well with AIX on the PowerPC and is a dramatic improvement over the Mach-based Rhapsody and MkLinux from Apple.

The trend in processor design seems to be directed towards hardware control of the MMU. Designers of the hardware may see this as a benefit to the OS developer but it is, in fact, a hindrance. Software control of the MMU allows experimentation with different allocation and storage strategies but hardware control of the MMU is too inflexible. As a final note, the architects of the PowerPC series seem to have decided to increase hardware control of memory management. Our results indicate that they might better spend their transistors and expensive silicon real-estate elsewhere.

References

[1] *Personal Communication with Steve Jobs.* 1998.

[2] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice Hall, 1992.

[3] Jochen Liedtke. The performance of μ-kernel-based systems. In *Proceedings of SOSP '97*, 1997.

[4] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, 1992.

[5] Larry McVoy. lmbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference.* Usenix, 1996.

[6] Motorola and IBM. *PowerPC 603 User's Manual.* Motorola, 1994.

[7] Theodore Romer, Wayne, Ohlrich, Anna Karlin, and Brian Bershad. Reducing tlb and memory overhead using online superpage promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[8] Ed Silha. *The PowerPC Architecture, IBM RISC System/6000 Technology, Volume II.* IBM Corp., 1993.

[9] Richard L. Sits. Alpha axp architecture. *Communications of the ACM*, February 1993.

[10] Mark Swanson, Leigh Stoller, and John Carter. Increasing tlb reach using superpages backed by shadow memory. In *Computer Architecture News*, 1998.

[11] Madhusudhan Talluri, Mark D. Hill, and Yousef A. Khalidi. A new page table for 64-bit address space. In *Proceedings of SOSP '95*, 1995.

[12] Shreekant S. Thakkar and Alan E. Knowles. A high-performance memory management scheme. *IEEE Computer*, May 1986.

Logical vs. Physical File System Backup

Norman C. Hutchinson
Computer Science Department
University of British Columbia
Vancouver, B.C., Canada V6T 1Z4
hutchinson@cs.ubc.ca, http://www.cs.ubc.ca/spider/norm

Stephen Manley, Mike Federwisch, Guy Harris
Dave Hitz, Steven Kleiman, Sean O'Malley
Network Appliance, Inc.
2770 San Tomas Expressway
Santa Clara, CA 95051
{stephen,mikef,guy,hitz,srk,sean}@netapp.com, http://www.netapp.com

Abstract

As file systems grow in size, ensuring that data is safely stored becomes more and more difficult. Historically, file system backup strategies have focused on logical backup where files are written in their entirety to the backup media. An alternative is physical backup where the disk blocks that make up the file system are written to the backup media. This paper compares logical and physical backup strategies in large file systems. We discuss the advantages and disadvantages of the two approaches, and conclude by showing that while both can achieve good performance, physical backup and restore can achieve much higher throughput while consuming less CPU. In addition, physical backup and restore is much more capable of scaling its performance as more devices are added to a system.

1 Introduction

As a central player in every operating system, file systems have received a tremendous amount of attention from both the academic and industrial communities. However, one important aspect of file systems has been conspicuous by its absence in the literature - file system backup and restoration. As file systems grow in size, ensuring that data is safely stored becomes more and more difficult. The research literature on this topic is extremely limited; Chervenak et al. present a good survey of the current literature [CVK98]. On the other hand, industry is very interested in both the correctness and performance of file system backup [Sun97, CCC98].

In evaluating a backup/restore strategy, a number of end-user desires must be balanced. On the backup side, it is the hope of every system administrator to never need any of the backup data that is collected, and in fact, in normal operation the vast majority of the data that is backed up is never looked at. This means that maximizing the speed with which data can be backed up, and minimizing the resources (disk and CPU) that are used in performing the backup are very important. This also means that the robustness of backup is critical. Horror stories abound concerning system administrators attempting to restore file systems after a disaster occurs, only to discover that all the backup tapes made in the last year are not readable.

Because backup data is kept for a long time, both to provide file system history and increased resilience to disasters, it is important that the format used to store the data be archival in nature. That is, the data should still be recoverable even if the hardware, operating system, or backup/restore software of the system has been changed since the backup was created.

On the restore side, there are at least two primary sorts of restores that are performed. We call these disaster recovery and stupidity recovery. Disaster

recovery comes into play when whole file systems are lost because of hardware, media, or software failure. A disaster recovery solution involves a complete restore of data onto new, or newly initialized media. The solution to the disaster recovery requirement also allows migration of data from one set of media to another. Stupidity recovery manifests itself as requests to recover a small set of files that have been "accidentally" deleted or overwritten, usually by user error.

There are two primary approaches to the backup/restore problem: logical and physical. A logical (or file-based) strategy interprets the file system meta data, discovers which files need to be duplicated and writes them to the backup media, usually in a canonical representation which can be understood without knowing very much if anything about the file system structure. A dump command has been implemented in every version of Unix since Version 6 from AT&T. The current standard was developed as part of the original BSD Unix effort. Other logical backup approaches include using tar or cpio. Each of these tools define their own format for the data, but in both cases the format is architecture neutral and well documented. A number of companies such as Legato, Veritas, and IBM have extended this idea by defining their own proprietary formats (typically based on tar or cpio) which can be used to stream file system data to a backup server.

A physical (or block-based) strategy duplicates the physical medium on which the files are stored (disks or disk arrays) onto the backup medium without interpretation (or with a minimum of interpretation). Physical backup has primarily been used to copy data from one medium to another (the Unix "dd" command), but has also been developed into a fully functional backup/restore strategy by Digital [GBD96]. The Plan 9 file system uses a physical block level copy-on-write scheme to implement its file system "epoch" scheme, which is similar in some respects to incremental file system backup [Qui91]. One of the advantages of a physical dump is that all file and file system attributes are duplicated, even those that may not be representable in the standard archival format. Examples of such attributes include CIFS access control lists, snapshots, hidden files, file system configuration or tuning information and file system statistics.

Recent years have witnessed a quiet debate between the merits of file-based versus block based backup schemes. Unfortunately, the comparisons have generated little interest for two reasons. First, since the two schemes are fundamentally different, it is difficult to find common ground on which to base reasonable analyses. Second, it is rare to find systems in which the two schemes have both been implemented with comparable degrees of completeness and attention to detail.

Network Appliance's WAFL (Write Anywhere File Layout) filesystem [HLM94] implements both the logical and physical backup strategies. By using snapshots (consistent, read-only images of the file system at an instant of time) both logical and physical dump can backup a consistent picture of the file system. WAFL's physical backup strategy is called image dump/restore. It takes advantage of the snapshot implementation to quickly find those disk blocks that contain data that needs to be dumped. Furthermore, the bookkeeping necessary to support copy-on-write enables incremental image dumps — only those disk blocks that have changed since the last image dump are included in an incremental dump. While taking advantage of the benefits of WAFL, image dump bypasses many of the file system constructs when reading and restoring data in order to improve performance. Thus, it is a true block-based implementation.

WAFL's BSD-style dump and restore utility has also been modified to take advantages of the features of the WAFL file system. First and foremost, unlike most other BSD-style dumps, the Network Appliance dump is built into the kernel. Since Network Appliance's filers are specialized to the task of serving files, there is no user level. Instead the file system has been designed to include dump and restore. This not only avoids context switches and data copies, but further allows dump and restore to utilize their own file system access policies and algorithms as well as giving them access to internal data structures. Unlike the image dump utility which bypasses the filesystem however, BSD-style dump and restore uses WAFL to access data.

WAFL therefore provides an intriguing test-bed for comparing and contrasting file-based and block-based backup strategies. First, rarely is a file system designed with backup as one of the primary goals. Second, system designers do not usually optimize both block-based and file-based backup. Finally, the nature of WAFL enables functionality that supersedes fundamental backup and restore. On the physical backup side, WAFL's image dump technol-

ogy allows more interesting replication and mirroring possibilities. On the logical backup side, some companies are using dump/restore to implement a kind of makeshift Hierarchical Storage Management (HSM) system where high performance RAID systems nightly replicate data on lower cost backup file servers, which eventually backup data to tape.

Section 2 of this paper describes those features of WAFL that are important in our discussion of backup strategies. Section 3 of this paper describes logical dump as embodied in WAFL's modified BSD dump. Section 4 describes WAFL's physical dump strategy (image dump). Section 5 compares the performance of the two utilities. Section 6 describes the future possibilities inherent in each scheme, and Section 7 concludes.

2 WAFL - Write Anywhere File Layout

In many respects, WAFL's disk format is similar to that of other UNIX file systems such as the Berkeley Fast File System [MJLF84] and TransArc's Episode [CAK+92] file system. For example:

- WAFL is block based, using 4 KB blocks with no fragments.
- WAFL uses inodes to describe its files.
- Directories are specially formatted files.

Like Episode, WAFL uses files to store meta-data. WAFL's two most important meta-data files are the inode file (which contains all inodes) and the free block bitmap file. Keeping meta-data in files allows meta-data blocks to be written anywhere on disk. WAFL has complete flexibility in its write allocation policies because no blocks are permanently assigned to fixed disk locations as they are in the Berkeley Fast File System (FFS). The only exception to the write anywhere policy is that one inode (in WAFL's case the inode describing the inode file) must be written in a fixed location in order to enable the system to find everything else. Naturally, this inode is written redundantly.

2.1 The Snapshot Facility

Snapshots are a primary benefit of WAFL's write anywhere approach. A snapshot is an on-line, read-only copy of the entire file system. The file server is able to copy the entire file system in just a few seconds. The copy uses no additional disk space until files are changed or deleted due to the use of copy-on-write. Only as blocks in the active file system are modified and written to new locations on disk does the snapshot begin to consume extra space.

Snapshots can be used as an on-line backup capability allowing users to recover their own files. Snapshots can be taken manually, and are also taken on a schedule selected by the file system administrator; a common schedule is hourly snapshots taken every 4 hours throughout the day and kept for 24 hours plus daily snapshots taken every night at midnight and kept for 2 days. With such a frequent snapshot schedule, snapshots provide much more protection from accidental deletion than is provided by daily incremental backups. WAFL allows up to 20 snapshots to be kept at a time. How long snapshots can be kept depends on how quickly the file system changes, but ranges from a few hours to a few weeks. Snapshots also simplify backup to tape. Since snapshots are read-only copies of the entire file system, they allow self-consistent backup from an active system. Instead of taking the system off-line, the system administrator can make a backup to tape of a recently created snapshot.

A WAFL file system can be thought of as a tree of blocks rooted by a data structure that describes the inode file. The inode file in turn contains the inodes that describe the rest of the files in the system including meta-data files such as the free block bitmap. WAFL creates a new snapshot by making a duplicate copy of the root data structure, and updating the block allocation information. Since the root data structure is only 128 bytes, and since only the block allocation information needs to be copied on disk, a new snapshot does not consume significant additional disk space until a user deletes or modifies data in the active file system. WAFL creates a snapshot in just a few seconds.

Due to the presence of snapshots, the question of whether a block is free or in use is more complicated than in file systems without snapshots. Typically, when a file is deleted all of the blocks holding data for that file may be marked as free. In WAFL,

however, if the file is in a snapshot, then the blocks must not be reused until all of the snapshots holding the file are deleted. Accordingly, WAFL's free block data structure contains 32 bits per block in the file system instead of the 1 bit per block that is needed without snapshots. The live file system as well as each snapshot is allocated a bit plane in the free block data structure; a block is free only when it is not marked as belonging to either the live file system or any snapshot.

2.2 Consistency Points and NVRAM

At least once every 10 seconds WAFL generates an internal snapshot called a consistency point so that the disks contain a completely self-consistent version of the file system. When the filer boots, WAFL always uses the most recent consistency point on disk, which means that even after power loss or system failure there is no need for time consuming file system checks. The filer boots in just a minute or two, most of which is spent spinning up disk drives and checking system memory.

The filer uses non-volatile RAM (NVRAM) to avoid losing any NFS requests that might have occurred after the most recent consistency point. During a normal system shutdown, the filer turns off NFS service, flushes all cached operations to disk and turns off the NVRAM. When the filer restarts after a system failure or power loss, it replays any NFS requests in the NVRAM that have not reached disk.

Using NVRAM to store a log of uncommitted requests is very different from using NVRAM as a disk cache, as some UNIX products do [LS89]. When NVRAM is used at the disk layer, it may contain data that is critical to file system consistency. If the NVRAM fails, the file system may become inconsistent in ways that fsck cannot correct. WAFL uses NVRAM only to store recent NFS operations. If the filer's NVRAM fails, the WAFL file system is still completely self consistent; the only damage is that a few seconds worth of NFS operations may be lost.

3 Logical Backup

The BSD dump utility has been available in various forms for nearly twenty years. In that time, the format has remained fairly constant, and the utility has been ported to platforms ranging from Solaris to Linux. One of the benefits of the format has been the ability to cross-restore BSD dump tapes from one system to another. Even if the utility did not already exist on the platform, such a utility could be written or ported since the format is publicly known.

The dump format is inode based, which is the fundamental difference between dump and tar or cpio. The dump format requires that all directories precede all files in the backup. Both directories and files are written in ascending inode order, assuming that inode #2 is the root of dump, not necessarily of the original file system. Directories are written in a simple, known format of the file name followed by the inode number. Each file and directory is prefixed with 1KB of header meta-data. This data includes: file type, size, permissions, group, owner, and a map of the 1KB holes in the file. The tape itself is prefixed with two bitmaps describing the system being dumped. The first map indicates which inodes were in use in the dumped subtree at the time of the dump. The second map indicates which inodes have been written to the backup media. The first map helps show which files have been deleted between incremental dumps. The second map verifies the correctness of the restore.

The format leads to a fairly simple dump utility. The dump runs as a four phase operation. A user specifies the incremental level of a dump of some subset of the file system. Since dump is file based, the incremental dump backs up a file if it has changed since the previously recorded backup - the incremental's base. A standard dump incremental scheme begins at level 0 and extends to level 9.

Once the dump level and path have been selected, the dump enters Phase I. This phase is basically a tree walk, marking the map with the used and to-be-dumped file inodes. Phase II then marks the directories between the root of the dump and the files selected in Phase I. These directories are needed for restore to map between file names and inode numbers. The final two phases of dump write out the directories and files in inode order, respectively.

Restore reads the directories from tape into one large file. It uses this to create a desiccated file system. That is to say, it tracks the offsets of the beginning of each directory in this file. So, when a user asks for a file, it can execute its own namei (that part of the kernel that maps file names to inodes), without ever laying this directory structure on the file system. This saves quite a bit of space. The reasoning behind this procedure revolves around the ability of restore to reclaim a subset of the data that is on tape. Thus, to write the full directory structure on the system is a bad idea, especially if it is low on disk space or inodes. Restore calculates which files need to be extracted. It then begins to lay the directories and files on the system. After the directories and files have been written to disk, the system begins to restore the directories' permissions and times. Note that it couldn't do this when creating the directories, since creating the files might have failed due to permission problems and definitely would have affected the times.

The primary benefits of a logical backup and restore utility can be traced to the use of the underlying file system. A user can back up a subset of a data in a file system, which can save time and backup media space. Furthermore, logical backup schemes often take advantage of filters - excluding certain files from being backed up. Again, one saves media space and time. Logical backups enable fairly easy stupidity recoveries. If a user accidentally deletes a file, a logical restore can locate the file on tape, and restore only that file. A logical backup is extremely resilient to minor corruption of the tape, or mistakes on the part of the backup utility. Since each file is self-contained, a minor tape corruption will usually affect only that single file. A mistake by the backup utility may result in some number of files being omitted, but much of the system should still be accessible on the tape. Since each file's data is grouped together, a logical backup stream tends to presuppose little or no knowledge of the source file system.

Not surprisingly, the weaknesses of a logical backup and restore utility can be traced to the use of the underlying file system. For basic operations, the utilities must use the file system, adding additional overhead. The file system's read-ahead and caching policies may not be optimized for backup. Concurrently, the backup's use of the file system may have serious impact on other users. Performance is often a very serious issue. The same concerns apply to a restore. For each file, metadata and data need to be written. Most file systems are not optimized for the data access patterns of a restore. Often, then we see the performance being traded for functionality. In some cases, the new functionality attempts to allay the performance problems. It is more often than not, unsuccessful.

Numerous formats and programs exist within the space of logical backup and recover utilities. The BSD dump utility, an inode based program that produces a commonly known output stream garners the advantages and weaknesses inherent in its design. The benefit of any well-known format is that the data on a tape can usually be easily restored on a different platform than that on which it was dumped. If a "dump" utility does not exist on the platform of choice, one can always port a different platform's dump. While certain attributes may not map across the different file systems, the data itself should be sound. Dump's advantages over other well-known formats, such as tar and cpio come from the ease of running incremental backups. Since dump maps inodes, each dump stores a picture of the file system as it looked at the time of the dump. Thus, an incremental dump not only saves those files that have changed, but also easily notes files that have been moved or deleted.

The weaknesses of dump arise from the inherent nature of being a well-known, inode based format. To create an output stream in a well-known format requires a potentially expensive conversion of file system metadata into the standard format. Furthermore, features of the file system that are not accounted for in the well-known format must either be lost or hidden in the stream. As people continually extend a well-known format to incorporate proprietary structures, such extensions can eventually conflict. These conflicts can harm or eliminate cross-platform advantages. This weakness plagues all well-known formats. Dump is further weakened by not being a completely self-identifying format. While this weakness does not exhibit itself during a backup, the fact that BSD restore needs to create a map of file-system on tape significantly slows down data recovery. Furthermore, the complexity of the operation increases the probability of error, and the consumption of system resources. Finally, since dump reads files in inode order, rather than by directory, standard file system read-ahead policy may not help, and could even hinder dump performance. Unnecessary data will continually be prefetched and cached. Dump enables cross-platform restores and accurate incremental backups. However, for these

advantages it adds a great deal of complexity in restoring systems and, as always, a well-known format can rapidly turn into a variety of proprietary formats that no longer work together.

Unlike the versions of BSD dump that run on Solaris, BSD OS, or Linux, the Network Appliance dump is part of the system's kernel. Since the architecture of the system does not allow for a user-level, the system was designed with dump as an integral part of the microkernel. The primary benefit to this approach is performance. For example, other dump utilities cross the user-kernel boundary to read data from files, and then to write the data to the tapefile. Network Appliance has implemented a no-copy solution, in which data read from the file system is passed directly to the tape driver. Unix versions of dump are negatively affected by the file system's read-ahead policy. Network Appliance's dump generates its own read-ahead policy. Similar fast paths have been set up for restoring data. The Network Appliance version of restore enjoys significant performance improvements that are only possible because it is integrated into the kernel and runs as root. Most notable among these is that while the standard restore accesses files and directories by name, which requires it to be able to construct pathnames that the file system can parse, the Network Appliance version directly creates the file handle from the inode number which is stored in the dump stream. In addition, since it runs as root, it can set the permissions on directories correctly when they are created and does not need the final pass through the directories to set permissions that user-level restore programs need.

Functionally, Network Appliance's dump and restore also add benefit to the standard program. By using snapshots, dump can present a completely consistent view of the file system. This not only helps users, but it obviates many consistency checks used by BSD restore on other systems. This enhances performance and simplifies restores. As discussed before, companies tend to extend the format in ways to back up attributes not included in the basic model. For Network Appliance, these attributes include DOS names, DOS bits, DOS file times, and NT ACLs created on our multi-protocol file system. None of these extensions break the standard format.

Still, there are some difficulties that arise due to being integrated in the kernel. First, since there is no "user level" only a person with root access to the filer can run restore. This is a disadvantage compared to the standard dump, restore, and tar utilities on Unix. The filer also does not support the interactive restore option due to limitations that arise from integrating restore into the kernel. On the whole, however, the benefits make the Network Appliance dump and restore a more popular option with users than mounting the file system onto a Unix machine, and using a Unix version of dump and restore.

4 Physical Backup

In its simplest form, physical backup is the movement of all data from one raw device to another; in the context of file system backup the source devices are disks and the destination devices may include disk, CD-Rom, floppy, Zip drives, and of course, tape.

It is a straightforward extension to the simple physical backup described in the preceding paragraph to interpret the file system meta-data sufficiently to determine what disk blocks are in use and only back those up. All file systems must have some way of determining which blocks are free, and the backup procedure can interpret that information to only back up the blocks that are in use. Naturally, this requires that the block address of each block written to the backup medium be recorded so that restore can put the data back where it belongs.

The primary benefits of a physical backup scheme are simplicity and speed. It is simple because every bit from the source device is copied to the destination; the format of the data is irrelevant to the backup procedure. It is fast because it is able to order the accesses to the media in whatever way is most efficient. There are a number of limitations to physical backup, however. First, since the data is not interpreted when it is written, it is extremely non-portable; the backup data can only be used to recreate a file system if the layout of the file system on disk has not changed since the backup was taken. Depending on the file system organization, it may even be necessary to restore the file system to disks that are the same size and configuration as the originals. Second, restoring a subset of the file system (for example, a single file which was accidently deleted) is not very practical. The entire file system must be recreated before the individual disk blocks that make up the file being requested

can be identified. Third, the file system must not be changing when the backup is performed, otherwise the collection of disk blocks that are written to disk will likely not be internally consistent. Finally, the coarse grained nature behind this method leads to its own difficulties. Because file system information is not interpreted by the backup procedure, neither incremental backups nor backing up less than entire devices is possible. A raw device backup is analogous to a fire hose. Data flows from the device simply and rapidly — but it is all the you can do to hold the hose. Finer grained control is generally impossible.

These negative aspects of physical backup have until now caused it to have very limited application. Physical backup is used in a tool from BEI Corporation that addresses the problem of restoring NT systems after catastrophic failure of the system disk [Edw97]. Two large experiments at getting Terabyte per hour backup performance mention the use of raw (or device) backup and contain performance measures that validate the intuition that it obtains extremely good performance [CCC98, Sun97].

4.1 WAFL Image Dump

WAFL's snapshot facility addresses several of the problems with physical dump identified above. First, because a snapshot is a read-only instantaneous image of the file system, copying all of the blocks in a snapshot results in a consistent image of the file system being backed up. There is no need to take the live file system off line.

Second, snapshots allow the creation of incremental physical dumps. Since each snapshot has a bitmap indicating which blocks are in the snapshot, we can easily determine which blocks are new and need to be dumped by considering the sets of blocks included in the two snapshots. Suppose that our full backup was performed based on a snapshot called A, and we have just created a new snapshot called B from which we wish to make an incremental image dump. Table 1 indicates the state of blocks as indicated by the bits in the snapshot bit planes. We must include in the incremental dump every block that is marked as used in the bit plane corresponding to B but is not used the bit plane corresponding to A, and no other blocks. Alternatively, we can consider the bitmaps to define sets of blocks, we wish to dump the blocks in the set:

$$B - A$$

Higher level incrementals are handled in the same manner. A level 2 incremental whose snapshot is C and which is based on the full and level 1 incremental described above needs to include all blocks in:

$$C - B$$

since everything in A that is also in C is guaranteed to be in B as well. These sets are trivial to compute by looking at the bit planes associated with all of the relevant snapshots.

Of course, a block based dump does not want to be too closely linked to the internal file system, or you lose the advantage of running at device speed. Therefore, image dump uses the file system only to access the block map information, but bypasses the file system and writes and read directly through the internal software RAID subsystem. This also enables the image dump and restore to bypass the NVRAM on the file system, further enhancing performance.

Finally, the block based dump allows for some features that the file-based dump cannot provide. Unlike the logical dump, which preserves just the live file system, the block based device can backup all snapshots of the system. Therefore, the system you restore looks just like the system you dumped, snapshots and all. Unfortunately, the other two limitations detailed above, portability and single file restore, seem to be fundamental limitations of physical backup and are not addressed by WAFL. We'll return to the single file restore issue in Section 6.

5 Performance

This section reports the results of our experiments to quantify the achievable performance of the physical and logical backup and recovery strategies. We begin by reporting the performance of the most straightforward backup and recovery plans, and then proceed to measure the speedup achievable by utilizing multiple tape devices.

All of our measurements were performed on eliot, a Network Appliance F630 file server. The filer con-

Bit plane A	Bit plane B	Block state
0	0	not in either snapshot
0	1	newly written - include in incremental
1	0	deleted, no need to include
1	1	needed, but not changed since full dump

Table 1: Block states for incremental image dump

sists of a 500 MHz Digital Alpha 21164A processor with 512 MBytes of RAM and 32 MBytes of NVRAM. 56 9 GByte disks are attached via a Fibre Channel adapter. 4 DLT-7000 tape drives with Breece-Hill tape stackers are attached through dedicated Differential Fast Wide SCSI adapters. At the time the experiments were performed, the filer was otherwise idle.

The 481 GBytes of disk storage are organized into 2 volumes: home which contains 188 GBytes of data and rlse which contains 129 GBytes of data. The home volume consists of 31 disks organized into 3 raid groups and the rlse volume consists of 22 disks organized into 2 raid groups. Both the home and rlse file systems are copies (made using image dump/restore) of real file systems from Network Appliance's engineering department.

5.1 Basic Backup / Restore

Table 2 reports our measurements of backing up and restoring a large, mature[1] data set to a single DLT-7000 tape drive. In this configuration both logical and physical dump obtain similar performance, with physical dump getting about 20% higher throughput. This is because the tape device that we are using (a DLT-7000) is the bottleneck in the backup process. Note however the significant difference in the restore performance. This can be primarily attributed to image restore's ability to bypass the file system and write directly to the disk, while logical restore goes through the file system and NVRAM[2].

We can look at the resource utilization of the filer while backup and restore are proceeding to learn something about how logical and physical backup are likely to scale as we remove the bottleneck device. Table 3 indicate the time spent in various stages of backup and restore as well as the CPU utilization during each stage. It is worth noting the variation in CPU utilization between the two techniques. Logical dump consumes 5 times the CPU resources of its physical counterpart. Logical restore consumes more than 3 times the CPU that physical restore does.

One way to reduce the time taken by the backup procedure is to backup multiple file system volumes concurrently to separate tape drives. The resource requirements of both logical dump and physical dump are low enough that concurrent backups of the home and rlse volumes did not interfere with each other at all; each executed in exactly the same amount of time as they had when executing in isolation.

5.2 Backup and Restore to Multiple Tapes

Our next set of experiments are designed to measure how effectively each dump strategy can use parallel backup tape devices. For the logical strategy, we cannot use multiple tape devices in parallel for a single dump due to the strictly linear format that dump uses. Therefore we have separated the home volume into 4 equal sized independent pieces (we used quota trees, a Network Appliance construct for managing space utilization) and dumped them in parallel. For physical dump, we dumped the home volume to multiple tape devices in parallel. Tables 4 and 5 report our results.

Overall, logical dump managed to dump the 188 GByte home volume to 4 tape drives in 2.7 hours, achieving 69.6 GBytes/hour, or 17.4 GBytes/hour/tape. Physical backup to 4 drives completed in 1.7 hours, achieving 110 GBytes/hour, or 27.6 GBytes/hour/tape.

[1] A mature data set is typically slower to backup than a newly created one because of fragmentation: the blocks of a newly created file are less likely to be contiguously allocated in a mature file system where the free space is scattered throughout the disks.

[2] There is no inherent need for logical restore to go through NVRAM as it is simple to restart a restore which is interrupted by a crash. Modifying WAFL's logical restore to avoid NVRAM is in the works.

Operation	Elapsed time (hours)	MBytes/second	GBytes/hour
Logical Backup	7.4	7.2	25.0
Logical Restore	8.1	6.3	22.8
Physical Backup	6.2	8.6	30.3
Physical Restore	5.9	8.9	32.0

Table 2: Basic Backup and Restore Performance

Stage	Time spent	CPU Utilization
Logical Dump		
Creating snapshot	30 seconds	50%
Mapping files and directories	20 minutes	30%
Dumping directories	20 minutes	20%
Dumping files	6.75 hours	25%
Deleting snapshot	35 seconds	50%
Logical Restore		
Creating files	2 hours	30%
Filling in data	6 hours	40%
Physical Dump		
Creating snapshot	30 seconds	50%
Dumping blocks	6.2 hours	5%
Deleting snapshot	35 seconds	50%
Physical Restore		
Restoring blocks	5.9 hours	11%

Table 3: Dump and Restore Details

Operation	Elapsed time	CPU Utilization	Disk MB/s	Tape MB/s
Logical Backup				
Mapping	15 minutes	50%	5.0	0.0
Directories	15 minutes	40%	15.0	12.0
Files	4 hours	50%	12 - 20	12 - 20
Logical Restore				
Creating files	1.25 hours	53%	6.0	0.0
Filling in data	3.5 hours	75%	16.0	16.0
Physical Backup				
Dumping blocks	3.25 hours	12%	17	17
Physical Restore				
Restoring blocks	3.1 hours	21%	17.4	17.4

Table 4: Parallel Backup and Restore Performance on 2 tape drives

Operation	Elapsed time	CPU Utilization	Disk MB/s	Tape MB/s
Logical Backup				
Mapping	5 minutes	90%	9.5	0.0
Directories	7 minutes	90%	27.0	12.0
Files	2.5 hours	90%	21	21
Logical Restore				
Creating files	0.75 hours	53%	9.0	0.0
Filling in data	3.25 hours	100%	18.0	18.0
Physical Backup				
Dumping blocks	1.7 hours	30%	31	31
Physical Restore				
Restoring blocks	1.63 hours	41%	32	32

Table 5: Parallel Backup and Restore Performance on 4 tape drives

5.3 Summary

As expected, the simplicity of physical backup and restore means that they can achieve much higher throughput than logical backup and restore which are constantly interpreting and creating file system meta data. The performance of physical dump/restore scales very well; when physical restore hits a bottleneck it is simple to add addition tape drives to alleviate the bottleneck. Eventually the disk bandwidth will become a bottleneck, but since data is being read and written essentially sequentially, physical dump/restore allows the disks to achieve their optimal throughput. Logical dump/restore scales much more poorly. Looking at the performance of 4 parallel logical dumps to 4 tape drives (Figure 5) we see that during the writing files stage the CPU utilization is only 90% and the tape utilization is under 70% (as compared to that achieved by physical dump). The bottleneck in this case must be the disks. The essentially random order of the reads necessary to access files in their entirety achieves highly sub-optimal disk performance.

6 The Future

Physical backup and restore has been ignored for a long time. With file systems continuing to increase in size exponentially, the ability of a physical backup strategy to scale with increasing disk bandwidth makes it an interesting strategy. In the WAFL context, where snapshots provide the ability to discover recently changes blocks very efficiently, incremental image backups become a possibility. We are working on an implementation of incremental backup and will soon be able to answer questions about its performance. The image dump/restore technology also has potential application to remote mirroring and replication of volumes.

7 Conclusion

This paper has examined the limitations and performance of both the logical and physical backup and restore strategies. Logical backup has the advantage of a portable archival format and the ability to restore single files. Physical backup and restore has the advantages of simplicity, high throughput, and the ability to support a number of data replication strategies. Both have much to contribute to a complete file system backup strategy, but that the ability of physical backup/restore to effectively use the high bandwidths achievable when streaming data to and from disk argue that it should be the workhorse technology used to duplicate our constantly growing data sets and protect them from loss.

References

[CAK+92] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode file system. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 43–

60, Berkeley, CA, USA, January 1992. Usenix Association.

[CCC98] Sandeep Cariapa, Robert Clard, and Bill Cox. Origin2000 one-terabyte per hour backup white paper. SGI White Paper, 1998. http://www.sgi.com/Technology/teraback/teraback.html.

[CVK98] Ann L. Chervenak, Vivekenand Vellanki, and Zachary Kurmas. Protecting file systems: A survey of backup techniques. In *Proceedings of the Joint NASA and IEEE Mass Storage Conference*, March 1998.

[Edw97] Morgan Edwards. Image backup and NT boot floppy disaster recovery. *Computer Technology Review*, pages 54–56, Summer 1997.

[GBD96] R. J. Green, A. C. Baird, and J. C. Davies. Designing a fast, on-line backup system for a log-structured file system. *Digital Technical Journal of Digital Equipment Corporation*, 8(2):32–45, October 1996.

[HLM94] Dave Hitz, James Lau, and Michael Malcolm. File system design for a file server appliance. In *Proceedings of the 1994 Winter USENIX Technical Conference*, pages 235–245, San Francisco, CA, January 1994. Usenix.

[LS89] Bob Lyon and Russel Sandberg. Breaking through the nfs performance barrier. *Sun Technical Journal*, 2(4):21–27, Autumn 1989.

[MJLF84] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[Qui91] Sean Quinlan. A cached worm file system. *Software—Practice and Experience*, 21(12):1289–1299, December 1991.

[Sun97] High-speed database backup on Sun systems. Sun Technical Brief, 1997.

The Design of a Multicast-based Distributed File System

Björn Grönvall, Assar Westerlund, and Stephen Pink
Swedish Institute of Computer Science and *Luleå University of Technology*
{bg, assar, steve}@sics.se

Abstract

JetFile is a distributed file system designed to support shared file access in a heterogenous environment such as the Internet. It uses multicast communication and optimistic strategies for synchronization and distribution.

JetFile relies on "peer-to-peer" communication over multicast channels. Most of the traditional file server responsibilities have been decentralized. In particular, the more heavyweight operations such as serving file data and attributes are, in our system, the responsibility of the clients. Some functions such as serializing file updates are still centralized in JetFile. Since serialization is a relatively lightweight operation in our system, serialization is expected to have only minor impact on scalability.

We have implemented parts of the JetFile design and have measured its performance over a local-area network and an emulated wide-area network. Our measurements indicate that, using a standard benchmark, JetFile performance is comparable to that of local-disk based file systems. This means it is considerably faster than commonly used distributed file systems such as NFS and AFS.

1 Introduction

JetFile [15] is a distributed file system designed for a heterogenous environment such as the Internet. A goal of the system is to provide ubiquitous distributed file access and centralized backup functions without incurring significant performance penalties.

JetFile should be viewed as an alternative to local file systems, one that also provides for distributed access. It is assumed that, if a user trusts the local disk to be sufficiently available to provide file access, then JetFile should be available enough too. Ideally a user should have no incentive to put her files on a local file system rather than on JetFile.

JetFile is targeting for the demands of personal computing. It is designed to efficiently handle the daily tasks of an "ordinary" user such as mail processing, document preparation, information retrieval, and programming.

Increasing read and write throughput beyond that of the local disk has not been a goal. We believe that most users find the performance of local-disk based file systems satisfactory. Our measurements will show that it is possible to build a distributed file system with performance characteristics similar to that of local file systems.

A paper by Wang and Anderson [37] identifies a number of challenges that a geographically dispersed file system faces. We reformulate these slightly differently as:

Availability: Communication failures can lead to a file not being available for reading or writing. As a system grows, the likelihood of communication failures between hosts increases.

Latency: Latencies are introduced by propagation delays, bandwidth limitations, and packet loss. The special case, network disconnection, can be regarded as either infinite propagation delay or guaranteed packet loss.

Bandwidth: To reduce cost, it is in general desirable to keep communication to a minimum. Replacing backbone traffic with local traffic is also desirable because of its lower pricing.

Scalability: Server processing and state should grow slowly with the total number of hosts, files, and bytes.

JetFile is designed to use four distinct but complementary mechanisms to address these problems.

Optimistic algorithms are used to increase update availability and to reduce update latency.

Hoarding and prefetching will be used to increase read availability and to reduce read latency.

Replication and multicast are used to increase read availability, reduce read latency, reduce backbone traffic, and to improve scalability.

Clients act as servers to decrease update latency, minimize traffic, and to improve scalability.

Latencies introduced by the network can often be large, especially over satellite or wireless networks. This is an artifact of the limited speed of light and/or that packets must be repeatedly retransmitted before they reach their destination.

To hide the effects of network latencies, JetFile takes an optimistic approach to file updates. JetFile promises to detect and report update conflicts, but only after the fact that they have occurred.

The optimistic approach assumes that write sharing will be rare and uses this fact to hide network latencies. Experience from Coda [23, 19] and Ficus [29] tell us that in their respective environments write sharing and update conflicts were rare and could usually be handled automatically and transparently.

JetFile is designed to support large caches (on the order of gigabytes) to improve availability and to avoid the effects of transmission delays.

JetFile takes a "best-effort" approach with worst-case guarantees to maintain cache coherency. Coherency is maintained through a *lease* [14] or *callback* [17]-style mechanism[1]. The *callback* mechanism is best-effort in the sense that there is a high probability, but no absolute guarantee, that a *callback* reaches all destinations. If there are nodes that did not receive a *callback* message, the amount of time they will continue to access stale data is limited by a worst-case bound. In the worst case, when packet loss is high, JetFile provides consistency at about the same level as NFS [30]. Under more normal network conditions, with low packet drop rates, cache consistency in JetFile is much stronger and closer to that of the Andrew File System [17].

Applications that require consistency stronger than what JetFile guarantee may experience problems. This is an effect of the optimistic and best-effort algorithms that JetFile uses. Only under "good" network conditions will JetFile provide consistency stronger than NFS. To exemplify this, if a distributed make fails when run over NFS, it will sometimes also fail when run over JetFile.

There is a scalability problem with conventional *callback* and *lease* schemes: servers are required to track the identity of caching hosts. If the number of clients is large, then considerable amounts of memory are consumed at the server. With our multicast approach, servers need not keep individual client state since callbacks find their way to the clients using multicast. Rather than centralizing/concentrating the *callback* state at one server, the state has been distributed over a number of multicast routers. As an additional benefit, only one packet has to be sent when issuing the *callback*, not one for each host.

Traditional distributed file systems are often based on a centralized server design and unicast communication. JetFile instead relies on peer-to-peer communication over multicast channels. Most of the traditional file server responsibilities have been decentralized. In particular, the more heavyweight operations such as serving file data and attributes are, in our system, the responsibility of the clients. Some functions such as serializing file updates are still centralized in JetFile. Since serialization is a relatively lightweight operation in our system, serialization is expected to only have a minor impact on scalability.

Scalability is achieved by turning every client into a server for the files accessed. As a result of clients taking over server responsibilities, there is no need to immediately *write-through*[2] data to some server after a file update.

Shared files such as system binaries, news, and web pages are automatically replicated where they are accessed. Replication is used as a means to localize traffic, distribute load, decrease network round-trip delays, and increase availability. Replicated files are retrieved from a nearby location when possible.

Files that are shared, will, after an update, be distributed directly to the replication or *via* other replication sites rather than being transferred *via* some other server.

For availability and backup reasons, updated files will also be replicated on a storage server a few hours after update or when the user "logs off." The storage server thus acts as an auxiliary site for the file.

JetFile is designed to reduce the amount of network traffic to what is necessary to service compulsory cache misses and maintain cache coherency. Avoiding network communication is often the most efficient way to hide the effects of propagation delays, bandwidth limitations, and transmission errors.

We have built a JetFile prototype that includes most of JetFile's key features. The prototype does not yet have a storage server nor does it include any security related features. However, the prototype is operational enough for preliminary measurements. We have made measurements that will show JetFile's performance to be similar to local-disk based file systems.

The rest of this paper is organized as follows:

The paper starts with a JetFile specific tutorial on IP multicast and reliable multicast. The tutorial is followed by a system overview. The sections to follow are: File Versioning, Current Table, JetFile Protocol, Implementation, Measurements, Related Work, Future Work, Open Issues and Limitations, and Conclusions.

[1] A *callback* is a notification sent to inform that a cached item is no longer valid.

[2] The file is still written to local disk with the *sync* policy of the local file system.

2 Multicast

Traditionally multicast communication has been used to transmit data, often stream-oriented such as video and audio, to a number of receivers. Multicast is however not restricted to these types of applications. The inherent location-transparency of multicast also makes its use attractive for peer to multi-peer communication and resource location and retrieval. With multicast communication it is possible to implement distributed systems without any explicit need to know the precise location of data. Instead, peers find each other by communicating over agreed upon communication channels. To find a particular data item, it is sufficient to make a request for the data on the agreed upon multicast channel and any node that holds a replica of the data item may respond to the request. This property makes multicast communication an excellent choice for building a system that replicates data.

Multicast communication can also be used to save bandwidth when several hosts are interested in the same data by "snooping" the data as it passes by. For instance after a shared file is updated and subsequently requested by some host, it is possible for other hosts to "snoop" the file data as it is transferred over the network.

2.1 IP Multicast

In IP multicast [9, 8] there are 2^{28} (2^{112} in IPv6) distinct multicast channels. Channels are named with IP addresses from a subset of the IP address space. In this paper, we will interchangeably use the terms multicast address, multicast channel, and multicast group.

To multicast a packet, the sender uses the name of the multicast channel as the IP destination address. The sender only sends the packet once, even when there are thousands of receivers. The sender needs no knowledge of receiver-group membership to be able to send a packet.

Multicast routers forward packets along distribution trees, replicating packets as trees branch. Like unicast packets, multicast packets are only delivered on a best-effort basis. I.e., packets will sometimes be delivered in a different order than they were sent, at other times; they may not be delivered at all.

Multicast routing protocols [11, 3, 10, 24] are used to establish distribution trees that only lead to networks with receivers. Hosts signal their interest in a particular multicast channel by sending an IGMP [12] *membership report* to their local router. This operation will graft the host's local network onto, or prevent the host's local network being pruned from, the multicast distribution tree.

The establishment of distribution trees is highly dependent on the multicast routing protocol in use. Multicast routing protocols can roughly be divided into two classes. Those that create source specific trees and those that create shared trees. Furthermore, shared trees can be either uni- or bi-directional.

We will briefly touch upon one multicast routing protocol: Core Based Trees (CBT) [3]. CBT builds shared bi-directional trees that are rooted at routers designated to act as the "core" for a particular multicast group. When a leaf network decides to join a multicast group, the router sends a message in the direction towards the core. As the message is received by the next hop router, the router takes notice of this and continues to forward the message towards the core. This process will continue until the message eventually reaches the distribution tree. At this point, the process changes direction back towards the initiating router. For each hop, a new (hop long) branch is added to the tree. Each router must for each active multicast group keep a list of those interfaces that have branches. Thus, CBT state scales $\mathcal{O}(G)$, where G is the number of active groups that have this router on the path towards this groups core (see [5] for a detailed analysis). Remember that different groups can have different cores.

To make IP multicast scalable, it is not required to maintain any knowledge of any individual members of the multicast group, nor of any senders. Group membership is aggregated on a subnetwork basis from the leaves towards the root of the distribution tree.

In a way, IP multicast routing can be regarded as a network level filter for unwanted traffic. At the level of the local subnetwork, network adaptors are configured to filter out local multicasts. Adaptor filters protect the operating system from being interrupted by unwanted multicast traffic and allow the host to spend its cycles on application processing rather than packet filtering.

2.2 Scalable Reliable Multicast

IP packets are only delivered with best-effort. For this reason, JetFile communication relies on the Scalable Reliable Multicast (SRM) [13] paradigm. SRM is designed to meet only the minimal definition of reliable multicast, i.e, eventual delivery of all data to all group members. As opposed to ISIS [6], SRM does not enforce any particular delivery order. Delivery order is to some extent orthogonal to reliable delivery and can instead be enforced *on top of* SRM.

SRM is logically layered above IP multicast and also relies on the same lightweight delivery model. To be scalable, it does not make use of any negative or positive packet acknowledgments, nor does it keep any knowledge of receiver-group membership.

SRM stems from the Application Level Framing (ALF) [7] principle. ALF is a design principle where

protocols directly deal with application-defined aggregates suitable to fit into packets. These aggregates are commonly referred to as Application Data Units, or ADUs. An ADU is designed to be the smallest unit that an application can process out of order. Thus, it is the unit of error recovery and retransmission.

The contents of an ADU is arbitrary. It may contain active data such as an operation to be performed, or passive data such as file attributes. In the SRM context, ADUs always have persistent names. The name is assumed to always refer to the same data. To allow for changing data such as changing files, a version number is typically attached to the ADU's name.

The SRM communication paradigm builds on two fundamental types of messages, the *request* and the *repair*. The request is similar to the first half of a remote procedure call in that it requests for a particular ADU to be (re)transmitted. The repair message is quite different from its RPC counterpart. Any node that is capable of responding to the request prepares to do so but first initializes a randomized timer. When the timer expires, the repair is sent. However, if another node responds earlier (all nodes listen on the multicast address) the timer is canceled to avoid sending a duplicate repair. By initializing timers based on round-trip time estimates, repairs can be made from nodes that are as close as possible to the requesting node.

During times of network congestion, hosts behind the point of congestion will sometimes miss repair messages. In this case, it is sufficient if only one of the hosts make a request and then the corresponding repair will repair the state at all hosts.

If the reason to make a request is triggered by an external event such as the detection of a lost packet, one must be careful not to flood the network with request messages. In this case, multiple requests should be suppressed using a similar technique as with multiple repair suppression.

In SRM, reliable delivery is designed to be receiver-driven and is achieved by having each receiver responsible for detecting lost ADUs and initiating repairs. A lost ADU is detected through a "gap" in the version number sequence. Note that this approach only leads to eventual reliable delivery. There is no way for the receiver to know the "current" version number. The problem of maintaining current version numbers must be addressed outside of SRM in an application specific fashion. Also, applications will often be able to recover after a period of packet loss by only requesting the current data. Thus, it is not always necessary to catch up on every missed ADU. In essence, it is not reliable delivery of packets that matters; importance lies in reliable data delivery.

It is easy to build systems that replicate data with SRM. In JetFile we use SRM and replication to increase availability and scalability. This comes at a low cost since peers can easily locate replicas while at the same time the number of messages exchanged will be kept to a minimum.

3 System Overview

JetFile is built from a small number of components that interact by multicasting SRM messages[3]. These are:

File manager The traditional "file system client" that also acts as a file server to other file managers.

Versioning server The part of the system where file updates are serialized. Update conflicts are detected and resolved by file managers.

Storage server A server responsible for the long term storage of files and backup functions.

Key server A server that stores and distributes cryptographic keys used for signing and encrypting file contents.

We have not yet implemented the storage and key servers. This paper describes the file manager, the versioning server, and the protocol they use to interact. The key server is briefly discussed in the Future Work section.

In the JetFile instantiation of SRM, files are named using FileIDs. A FileID is similar to a conventional inode number. Using a hash function, FileIDs are mapped onto the multicast address space. It is assumed that the range of this mapping is large enough so that simultaneously used files will have a low probability of colliding on the same multicast address.

As mentioned in section 2.2, SRM only provides for eventual reliable delivery. Any stronger reliability must be implemented outside of SRM. In JetFile, this problem is addressed with a mechanism called the current table (section 7.2). The current table implements an upper bound to how long a file manager will be using version numbers that are no longer current.

When a file is actively used or replicated at a file manager, the file manager must join the corresponding multicast group. This way the manager will see the request for the file and will be able to send the corresponding repairs.

When reading a file, SRM is first used to locate and retrieve a segment of the file. When the file has been located, the rest of the file contents is fetched from this location using unicast.

Attached to the FileID is a version number. When a file changes, the versioning server is requested to generate a new version number. The request for the new

[3] Where multicast is not used it will be explicitly expressed.

version number and the corresponding repair are both multicast over the file channel. Because both the request and the repair are sent over the file channel these messages will also act as best-effort *callbacks*.

After a file has been updated, it is not immediately committed. Instead, the file is left in a tentative state. Only when/if the file is needed at some other host will it be committed. A file that has not yet been committed can not be seen by other machines.

JetFile directories are stored in regular JetFile versioned files. Filename related operations are always performed locally by manipulating the directory contents. Thus, both file creation and deletion are local operations.

4 File Versioning

Unlike traditional Unix file systems, JetFile adopts a file versioning model to handle file updates. A file is conceptually a suite of versions representing the file's contents at different times. To ensure that a file version is always consistent, JetFile prevents programs running at different nodes from simultaneously updating the file through the use of separate file versions. A new version of a file is writable at precisely one host. If some other host is to update the same file, the system guarantees that it will be writing to a different version. To update a file, it is necessary to request a new version number. Version numbers are assigned by the versioning server which acts as a serialization point for file updates. Once a file manager has acquired a new version number, it is allowed to update the file until the file is committed. A file is not committed until it is replicated at some other node. Thus, the new version number act as an update token for the file. The token is relinquished as a side effect of sending the file over the network.

Write-through techniques are not necessary in JetFile. After a file is updated, there is no need to write the file data through the file cache and over the network since the file manager is now, by definition, acting as a server for the file. The file will be put onto the network only when a file manager explicitly requests it and only at this point is the update token relinquished. This is an important property because it offloads both servers and networks in the common case when the file is not actively shared. Moreover, it is very likely that the file will soon be overwritten. Baker et. al [2] reports that between 65% and 80% of all written files are deleted (or truncated to zero length) within 30 seconds. Furthermore, it is shown that between 70% and 95% of the written bytes are overwritten or deleted within 2 hours.

Because file versions are immutable, JetFile insures that file contents will not change as a result of an update at some other node. The file contents is always consistent (assuming that applications write consistent output). This is particularly important for executable files. If the system allows changing the instructions that are being executed, havoc will surely arise. Most distributed and indeed even some local file systems do not keep the necessary state to prevent this from occurring.

JetFile groups sequences of writes into one atomic file update by bracketing file writes with pairs of `open` and `close` system calls. This approach implies that it is impossible to simultaneously "write share" a file at different hosts. The limited form of write sharing that JetFile supports is commonly referred to as "sequential write sharing" and means that one writer has to close the file before another can open the file for update. In our experience, sequential write sharing does not seriously limit the usefulness of a file system. Sequential write sharing is also the semantics chosen by both NFS and AFS.

Because JetFile only supports atomic file updates there need to be no special protocol elements corresponding to individual `writes`. Only `reads` have corresponding protocol elements.

Requests for new version numbers are addressed to the versioning server but sent with multicast. In this way file managers are informed that the file is about to change and can mark the corresponding cache item as changing. In response to the request, the versioning server sends an SRM repair message. This repair message will act as an unreliable *callback*.

To hide the effects of transmission delays and errors, JetFile takes an optimistic approach to file updates. When a file is opened for writing, the application is allowed to progress while the file manager, in parallel, requests a new version number. This approach is necessary to hide the effects of propagation delays in the network. For instance, the round-trip time between Stockholm and Sydney is about 0.5 seconds. If one was forced to update files in synchrony with the server and could not write the file in parallel with the request for a new file version, file update rates would be limited to two files per second, which is almost unusable.

It is arguable that these kind of optimistic approaches are dangerous and should be avoided because of the potential update conflicts that may arise. There is however empirical evidence indicating that sequential write sharing is rare, Kistler [19] and Spasojevic et. al [33] instrumented AFS to record and compare the identities of users updating files and directories. Kistler found that over 99% of all file and directory updates were by the previous writer. Spasojevic found that over 99% of all directory modifications were by the previous writer. Spasojevic were only able to report on directory write sharing due to a bug in the statistics collection tools. It should be noted that in Kistler's study few users would

use more than one machine at a time and that thus cross-user sharing should be similar to cross-machine sharing. In the Spasojevic study it is unknown how users spread over the machines.

In the event that two applications unknowingly update a file simultaneously, conflicting updates are guaranteed to be assigned different version numbers. This fact is used to detect update conflicts: one of the file managers will receive an unexpected version number and will signal an update conflict[4]. We intend to resolve conflicts with *application specific resolvers* as is done in Coda [23] and Ficus [29]. Currently update conflicts are reported but the latest update "wins" and "shadows" the previous update. We have deferred conflict resolution as future work.

File manager caches are maintained in a least-recently-used fashion with the constraint that locally created files are not allowed to be removed from the cache unless they have been replicated at the storage server. It is the responsibility of the file manager that performs the update to make sure that modified files in some way get replicated at the storage server before they are removed. Updated files are allowed to be transferred to the storage server *via* other hosts and even using other protocols such as TCP. The important fact is that the file manager verifies that a replica exists at the storage server before the file is removed from the cache.

5 Current Table

Unlike the unicast *callbacks* used by AFS, the multicast *callbacks* in JetFile do not verify that they actually reach all of their destinations. For this reason, JetFile instead implements an upper bound on how long a file manager can unknowingly access stale data. The upper bound is implemented with the use of a *current table*. The current table conceptually contains a list of all files and their corresponding highest version numbers. The *lifetime* of the current table limits how long a client may access stale data in case callbacks did not get through. If a file manager for some reason does not notice that a file was assigned a new version number, this will at the latest be noticed at the reception of the next current table. The current table is produced by the versioning server and can always be consulted to give a version number that is "off" by at most *lifetime* seconds.

The file manager requests a new current table before the old one expires. Since the current table is distributed with SRM, the table is only transferred every *lifetime* seconds. This work does not impose much load on the versioning server. If some hosts did not receive the table when it was initially transmitted, it will be retransmitted with SRM. If a current table is retransmitted, it is important that the sender first decrement the *lifetime* by the amount of time the table was held locally. The use of *lifetimes* rather than absolute expiration dates has the advantage of not requiring synchronized clocks.

If the current table contained a list of all files, it would clearly be too large to be manageable. For this reason, the file name space has been divided into volumes [32] and the current table is produced on a per volume basis. *Lifetime* is currently fixed at 30 seconds but should probably adapt to whether the volume is changing.

The prototype current table consists of pairs of file and version numbers. The frequent special case when the version number is *one* is optimized to save space. Version *one* is assumed by default. We also expect that the current table can be compressed using delta encodings and using differences to prior versions of the current table. We see these optimizations as future work.

The combination of multicast best-effort *callbacks* and current tables is similar to leases [14]. One of the differences is that with current tables it is not necessary to renew individual leases as lease renewal is aggregated per-volume. Another important difference is that the versioning server is stateless with respect to what hosts were issued a lease. This allows for much larger and more aggressive caching. However, our scheme does not provide any absolute guarantees with its best-effort *callbacks*, as does traditional leases.

In the normal case, JetFile expects that either the request or response message for a new version number will act as a *callback* break. When this fails, consistency is not much worse than for NFS since we use the current table as a fallback mechanism. By avoiding state in the versioning server, it can serve a much larger number of file managers. This comes at the price of only slightly decreased consistency guarantees.

6 The JetFile Protocol

At the JetFile protocol level, files are identified by a file identifier (FileID) similar to a Unix inode number. The FileID is represented by a tuple (*organization*, *volume*, *file-number*). The *organization* field divides the FileID space so that different organizations can share files. This is similar to the cell concept in AFS and DFS. Grouping files into volumes [32] allows several versioning and storage servers to exist and share load within one organization. Each volume is served by precisely one versioning server.

Every FileID is mapped with a hash function into a IP multicast address (the *file address*). All communication related to a file is performed on that file's corre-

[4]Note that since update conflicts are not detected until after a file is closed, the process that caused the conflict may already be dead.

sponding address which thus acts as a shared communication channel. There is also a multicast address for each volume (the *volume address*): the current table is transferred over this channel.

The basic JetFile protocol is simple. Messages consist of a generic header and message type specific parameters, see figure 1. A simplified header consists of a FileID, file version number, and the message type. In requests, the version number zero is used to indicate that the latest version is requested.

org	vol	f-num	vers	msg-type	param...

Figure 1: JetFile message header

message type	parameters	...	
status-request			
status-repair	attributes		
data-request	file-offset	length	
data-repair	file-offset	length	data...
version-request	transaction id		
version-repair	transaction id		
wakeup			

Table 1: JetFile messages and parameters

To request the file attributes of a specific file version, the FileID and version number are put into a status-request. The attributes are returned in a status-repair. To request the current file attributes without knowing the current version number, zero is used as the version number.

Requests for new version numbers are handled slightly differently. Because of the non-idempotent nature of version number incrementation, the versioning server must insure that a repeated request does not increment the version number more than once. To prevent this from occurring, the request (and the repair) carry a transaction id that is used to match retransmitted requests.

File contents is retrieved using data-request and data-repair messages. The requested file segment is identified by offset and length. Retrieval of an entire file is however somewhat more involved than to retrieve only one segment. First, a segment of the file is requested using SRM, then, when we know one source of the file we can *unicast* SRM compatible data-requests to the source, and receive *unicast* data-repairs. A TCP like congestion window is used to avoid clogging intermediate links. A more precise description of this protocol is outside the scope of this paper.

The *wakeup message* is used to gain the attention of the versioning server and will be described in section 7.2. The list of messages types and their parameters is shown in table 1.

The JetFile file name space is hierarchical and makes FileIDs transparent to applications and users. As in many other Unix file systems, directories are stored in ordinary files. The file manager performs directory manipulations (such as the insertion of a new file name) and the translation from pathnames to FileIDs. Directory manipulations are implemented as ordinary file updates. Thus, JetFile does not require any protocol constructs to handle directory manipulations.

7 Implementation

We have implemented a prototype of the JetFile design for HP-UX 9.05. Implemented are the file manager (split between a kernel module and a user-level daemon) and the versioning server. The storage server, key server and security related features have not been implemented.

7.1 File Manager

The implementation of the file manager consists of a small module in the operating system kernel and a user-space daemon process. The daemon performs all the protocol processing and network communication as well as implements the semantics of the file system. The traditional approach would be to implement all of this code inside the kernel for performance reasons. That it was possible to split the file system implementation between a kernel module and a user-level daemon with reasonable performance was shown to work for AFS-like file systems in [34]. There, the authors describe the split implementation of the Coda MiniCache. We will confirm their results by showing that it is possible to implement an efficient distributed file system mostly in user-space. The advantages of a user-level implementation are improved debugging support, ease of implementation, and maintainability. Also, having kernel and user modules tends to isolate the operating system-specific and the file system-specific parts, making it easier to port to other Unix "dialects" and other operating systems.

The file manager caches JetFile files in a local file system (UFS). Only whole-file caching has been implemented. This is, however, an implementation limitation and not a protocol restriction.

7.1.1 Kernel Module

The kernel module implements a file system, a character pseudo-device driver, and a new system call. Communication with the user-space daemon is performed by sending events over the character device. Events are sent from user-space to update kernel data structures and to reschedule processes that have been waiting for data to

arrive. The kernel, on the other hand, sends events with requests for data to be inserted into the cache or when about to update read-only cache items. When possible, events are sent asynchronously to support concurrency. For obvious reasons, this is not possible after experiencing a cache read miss. As an optimization, the device driver also supports the notion of "piggy backed" events, i.e. it is possible to send a number of events in sequence to minimize the number of kernel/user-space crossings. The contents of files are not sent over the character device, only references (*file handles*) are transferred.

A new system call is implemented to allow the user-space daemon to access files directly by file handle. This is simpler and more efficient than accessing them by name with `open`.

A typical example of the communication that can arise is the following:

1. the kernel module experiences a cache miss while trying to read an attribute.
2. the kernel module sends an event to the daemon and blocks waiting for the reply.
3. the daemon reads the event, does whatever is necessary to retrieve that data, installs the data in the kernel cache by sending one or more piggy backed events. The last event wakes up the blocked process.

The kernel module also implements a new file system in the Virtual File System (VFS) switch, called YFS (Yet another File System). The VFS-switch in HP-UX is quite similar to the Sun Vnode [21] interface.

The kernel cache contains specialized vnodes used by YFS (called *ynodes*), these have been augmented with fields to cache Unix attributes, access rights, and low-level file identifiers. There is also a reference to a local file vnode that contains the actual file data. The YFS redirects `reads` and `writes` to this local file with almost no overhead.

For simplicity of implementation, the user-space daemon handles `lookup` operations and the result is cached in the kernel directory-name lookup cache (DNLC [30]). When a lookup operation misses in the DNLC, an event is sent to user-space with a request to update the cache.

Pathname translation is performed by the VFS on a component by component basis through consulting the DNLC. For each pathname component, the cached access rights in the ynode are consulted to verify that the user is indeed allowed to traverse the directory. Symbolic links are cached in the same way as file data. After access rights verification, the `readlink` operation is performed by reading the value from the contents of the local file vnode.

To summarize, YFS consists of a cache of actively and recently used ynodes and a cache of translations from directory and filenames to ynodes. The ynodes cache Unix attributes, a reference to a local vnode, and access rights. The local vnode contains file data, directory data, or the value of a symbolic link.

7.1.2 User-space File Manager

The responsibilities of the user-space file manager is to keep track of locally created and cached files. The file manager listens for messages on the multicast addresses of all cached files and for events from the kernel on the character device. When a message arrives from the network, the target file is looked up in the table of all cached files. If the file is found, the message is processed otherwise the message is discarded. These spurious messages result from collisions in the hash from FileID to multicast address or limitations in the host's multicast filtering.

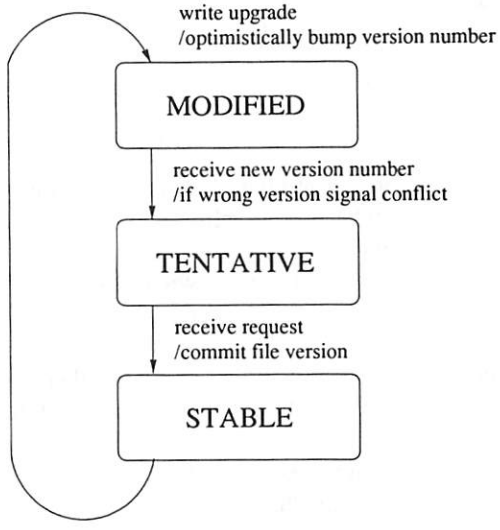


Figure 2: File states

For each cached file, the user-space file manager keeps track of the current version and of the file *state*. The most important states are shown in figure 2. Locally created files start in state *tentative*. The file will stay in this state and can be updated any number of times until some other file manager requests it, at this time the file is committed and the state changes to *stable*. Files received over the network always start in state *stable*.

If some application wants to modify a file that is *stable*, a request for a new version number is made. After sending the version-request, the file manager changes the state to *modified* and optimistically increments the version number by one. A file in the *modified* state only

exists locally and cannot be sent to other file managers. When the new version number arrives, the file state will change to *tentative*. If the version number received from the versioning server is not the expected one, the file manager reports that there was an update conflict.

7.2 Versioning Server

The versioning server runs entirely as a user-space daemon. It keeps track of the current version of all files in a volume and replies to requests for new version numbers. Every version number request carries a random transaction id that will be copied to the reply. This is to handle messages that get lost and have to be retransmitted.

The versioning server does not always listen on all file addresses in the managed volumes. Initially, it only listens on volume addresses. When a file manager does not get a repair back from a version-request, the file manager sends a *wakeup message* on the volume address prompting the versioning server to join the multicast group named by the file address. When creating files, the file manager by definition has a token for creating the first version. This avoids needless requests for initial version numbers.

Allocation of file numbers in a volume is handled by the versioning server to guarantee their uniqueness.

The versioning server is also responsible for creating and distributing the current table. The table consists of all files in the volume and their latest version numbers.

8 Measurements

The goal of these measurements is to show that the JetFile design has performance characteristics that are similar to existing local file systems when running with a warm cache.

To make the evaluation we used a standard distributed file systems benchmark, the Andrew Benchmark [16]. This benchmark operates on a set of files constituting the source code of a simple Unix application. Its input is a subtree of 71 files totaling 370 kilobytes distributed over 5 directories. The output consists of 20 additional directories and 91 more files. Thus, in this benchmark the file manager joins 187 IP multicast groups plus one for the current table.

The benchmark consists of five distinct phases: *MakeDir*, which constructs 5 subtrees identical in structure to the source subtree; *CopyAll*, which populates one of the target subtrees with the benchmark files; *StatAll*, which recursively "stats" every file at least once; *ReadAll*, which reads every file byte twice, and finally *Compile* which compiles and links all files into an application.

Tests were conducted over a 10Mb/s Ethernet on HP 735/99 model workstations. In table 2 we present averages and standard deviations over seven runs. We use one machine as the benchmark machine; another is used as a JetFile file manager housing the files, thus acting as a server. Finally, a third machine is used as a versioning server.

To emulate a wide area network (WAN) with long transmission delays, we put a bridge between the benchmark machine and the two other machines. The bridge was configured to delay packets to yield a round trip time of 0.5 seconds.

The benchmark was run first in the local HP-UX Unix File System (UFS) with a hot cache, i.e. all files were already in the buffer cache and file attributes were cached in the kernel. We then ran JetFile with a hot cache, and finally, we ran the same benchmark but with a cold JetFile cache on the benchmark machine so that all files first had to be retrieved over the network.

Before we start to make comparisons, a few facts need to be pointed out:

For unknown reasons the HP-UX UFS initially creates directories of length 24 bytes. When the first name is added, the directory length is extended to 1024 bytes. When adding more names, the directory is normally not extended, only updated. All this results in an extra disk block being written when adding the first name to a directory. JetFile directories start with a length of 2048 bytes and stays that long until full. In both UFS and JetFile, changes to directory blocks are always written synchronously to disk.

Inode allocation in JetFile is asynchronous and logged. The log records are written to disk when either a log disk block becomes full or when the file system is idle. In HP-UX UFS, inode allocation is always synchronous.

We first compare UFS with JetFile, both running with hot caches:

In the *MakeDir* phase, JetFile performs slightly better because UFS suffer from extending directories when adding the first name. JetFile also benefit from its asynchronous inode allocation.

In the *CopyAll* phase, JetFile benefit the most from its asynchronous inode allocation. Note that JetFile stores file data in local UFS files. Both UFS and JetFile use precisely the same "flush changes to disk every 30 seconds" algorithm. Only inode allocation differ between UFS and JetFile.

In the *StatAll*, *ReadAll*, and *Compile* phases, performance of JetFile and UFS are very similar.

Next we compare JetFile with hot and cold caches:

The only difference in the *MakeDir* phase is that we need to acquire a new version number for the top level directory. Since we optimistically continue to write directories while waiting for the new version number to arrive, we should expect only marginal differences in

Phase	UFS hot	JetFile hot	E-WAN hot	JetFile cold	E-WAN cold
MakeDir	1.55 (0.01)	1.22 (0.06)	1.26 (0.03)	1.25 (0.03)	1.28 (0.02)
CopyAll	2.68 (0.06)	1.56 (0.02)	1.55 (0.04)	3.71 (0.13)	50.86 (0.21)
StatAll	2.60 (0.02)	2.59 (0.01)	2.58 (0.01)	2.60 (0.01)	2.58 (0.01)
ReadAll	4.99 (0.02)	5.01 (0.02)	5.01 (0.02)	5.01 (0.02)	5.02 (0.05)
Compile	11.16 (0.05)	11.05 (0.03)	11.05 (0.07)	11.08 (0.08)	11.04 (0.07)
Sum	22.98 (0.08)	21.43 (0.04)	21.45 (0.07)	23.65 (0.12)	70.79 (0.28)

Table 2: Phases of the Andrew benchmark. Means of seven trials with standard deviations. JetFile over an emulated WAN (E-WAN) had a round trip time of 0.5 seconds. All times in seconds, smaller numbers are better.

elapsed time, regardless of round trip time.

In the *CopyAll* phase with a cold cache, the benchmark makes a copy of every file byte, while at the same time JetFile transfers all the files over the network and writes them to local disk. In effect, every byte is written to disk twice, although asynchronously. The combination of waiting for the file bytes to arrive so that the copying can continue in combination with issuing twice as many disk writes explains why the elapsed time increases from 1.56 to 3.71 seconds.

Running with a cold cache over the emulated WAN results in times for the *CopyAll* phase to increase from 3.71 to 50.86 seconds. This is to expect since it takes 0.5 seconds to retrieve a one byte file over the emulated WAN. There is no way out of this problem other than to fetch files before they are referenced. Prefetching will be briefly discussed in the Future Work section 10.2.

File sys.	Warm cache	Cold cache
UFS	22.98 (0.08) 100%	N/A
JetFile	21.43 (0.04) 93%	23.65 (0.12) 103%
AFS	26.49 (0.15) 115%	28.40 (0.60) 124%
NFS	29.54 (0.05) 129%	30.20 (0.20) 131%

Table 3: Total elapsed time of the Andrew benchmark. Percent numbers are normalized to UFS with a hot cache. All times in seconds, smaller numbers are better.

It is interesting to compare JetFile to existing commercially available distributed file systems. We repeated the above experiments using the same machines and network. One machine was used as the benchmark machine and a second was used either as an AFS or NFS file server. Even with tuned commercial implementations such as AFS and NFS, users pay a performance penalty on the order of 20% to make their files available over the network.

It would be very interesting to measure the scalability of the system. Unfortunately, we currently do not have a JetFile port to a more popular kind of machine.

9 Related Work

The idea of building a storage system around the distribution of immutable object versions is not entirely new. These ideas can be traced back to the SWALLOW [28] distributed data storage system. SWALLOW is, however, mostly concerned with the mechanisms necessary to support commitment synchronization between objects, a relatively heavyweight mechanism that is usually not required in distributed file systems.

Another system based on optimistic versioned concurrency control is the Amoeba distributed file system [26]. Their concurrency algorithms allows for simultaneous read and write access by verifying read and write sets before a file is allowed to be committed. Since JetFile targets an environment where files are usually updated in their entirety and write sharing is rare, we decided to use a more lightweight approach.

Coda uses the technique "trickle integration" [27] to reduce delay and bandwidth usage when updating files. File updates are written to the server as a background task after they have matured and been subject to write annulation optimizations. This should be contrasted with the JetFile approach where clients are turned into servers. The JetFile approach make file updates available earlier. The Coda people argue that this is not always desirable, a strongly connected client can be forced to wait for data propagation before it can continue to read the file.

AFS [17] and DFS [18] both implement a limited form of read-only replication. This replication is static and has to be manually configured to meet the expected load and availability. In JetFile the replication is dynamic and happens where the files are actually used rather than where they are expected to be used.

In xFS [1] servers have been almost eliminated by making all writes to distributed striped logs. The design philosophy is "anything anywhere" which should make the system scalable. This is however a different form of scalability: the goal is to scale the number of nodes within a compute-cluster and to improve throughput. The goal of JetFile is to scale geographically over different network technologies. JetFile is also self con-

figuring. There is no need to configure clients as servers or vice versa since JetFile clients per definition take server responsibilities for files accessed.

Frangipani [36] is also designed for compute-cluster scalability, but takes a different design approach. It is designed using a layered structure with files stored in Petal [25] (a distributed virtual disk) and complemented with a distributed lock service to synchronize access.

Both xFS and Frangipani suggest that protocols such as NFS, AFS, and DFS [18] be used to export the file system and provide for distributed access. As far as we know, nothing in our design prevents JetFile file managers from storing their files in xFS or Frangipani.

An earlier xFS paper [37] discusses a WAN protocol designed to connect several xFS clusters. Each cluster has a consistency server, and inter-cluster control traffic, flow through these. Like the JetFile protocol, this protocol also addresses typical WAN problems such as availability, latency, bandwidth, and scalability. This is done by a combination of moving file *ownership* (read or write) between clusters and on-demand-driven file transfers between clients. To reduce consistency server state, file *ownership* is aggregated on a directory subtree basis.

10 Future Work

10.1 Data Security and Privacy

The security architecture of JetFile is an important and central part of the entire design.

Validating the integrity of data is necessary in a global environment. It is also vital to be able to verify the originator of data to prevent impostors masquerading as originators. In JetFile, both these issues will be handled with the use of digital signatures [31].

To allow for cryptographic algorithms with different strengths, JetFile defines different types of keys. These key types in turn define signature types. For instance, there might be one key of type (MD5, RSA-512) which means that when this key is used to make signatures the algorithm MD5 should be used to generate 128 bit cryptographic checksums and RSA with 512 bit keys shall be used for encryption. A different and stronger key type is (SHA1, RSA-1024). Note that key types not only define the algorithms to use but also key lengths.

Since JetFile design is based on the Application Level Framing (ALF, section 2.2) principle it is most natural to protect application defined data units rather than the communication per se. The data units that make up a file are:

Data object: holds the file data and is only indirectly protected.

Status object: holds file attributes. It also contains a lists of cryptographic checksums, each sum protect a segment of the file. The status object is signed by the file author.

Protection object: holds a list of users that are allowed to write to the file and a reference to a privacy object. The protection object is signed by the system.

Privacy object: holds a list of users that are allowed to read the file. It also contains a key that is used to encrypt file contents when the file is transferred over the network.

To verify the integrity of a publicly readable file, first retrieve (with SRM) the key used to sign the status object, check that the author is allowed to write to the file by consulting the protection object and verify the signature. If everything is ok, verify the file segment checksums (checksum algorithm is derived from key type).

Verifying private files is only slightly different. Before the file segment checksums can be calculated the file data must be decrypted with the key that is stored in the privacy object.

The privacy object is not per file. Some files have no associated privacy object at all (publicly readable files). Private files can also be grouped together to share a common privacy object by all pointing out the same privacy object. Note that private files are always transferred encrypted over the network.

Privacy objects cannot be distributed with SRM and shall instead be transferred from the key server through encrypted channels (the channels that are used for bootstrapping, more about this below).

When a user "logs on" a workstation the necessary keys to make signatures are generated. The private key is held locally and is used to make signatures. The public key is registered with the key server using a bootstrap system such as Kerberos [35]. The key server will then sign this key and it will be distributed with SRM to all interested parties. The key used by the system for signing is given to the user during this initial bootstrap.

In JetFile signatures have limited lifetimes, i.e. an object that has been signed can only be sent over the network as long as the signature is still valid. There is no reason to discard an object from the local cache just because the signature expired. Expiration only means that if this object would be sent over the network, receivers will not accept it because the signature is no longer valid. The signature lifetime is derived from the key lifetime. When the key server publishes keys on behalf of users they are also marked with an issue and expiration date.

Signatures will eventually expire, there must be somewhere in the system to "upgrade" signatures. This task is assigned to the storage server. Files are initially signed

by users, then stabilize and migrate to the storage server, and eventually their signature expires. At this point, the storage server creates a new signature using a system key and the file is ready for redistribution.

Encryption algorithms are often CPU intensive, we intend to use keys with short key lengths for short lived files such as the current table. Similarly when files are updated they are first signed with a lightweight signature. If the file stabilizes, its signature gets upgraded.

Our strategy of migrating files to the server every few hours is intended to avoid needless CPU-consuming encryptions. In systems such as AFS and NFS where file updates are immediately written through to the server, CPU usage can be costly. In JetFile we only have to invoke the encryption routines when the file is committed, and this happens only when the file is in fact shared, or when it must be replicated at the storage server.

If all files are both secret and shared then there will be extensive encryption. We expect, however, only a small percentage of files to fall into this category and accept that some encryption overhead is unavoidable. In normal operation, users do not share their secret files, so encryption only has to be performed on transfer to and from the storage server.

10.2 Hoarding and Prefetching

For JetFile to work well in a heterogenous environment it must be prepared to handle periods of massive packet loss and even to operate disconnected from the network. JetFile's optimistic approach to handle file updates is only one aspect of attacking these problems. There must also be mechanisms that try to avoid compulsory cache misses.

One way of avoiding compulsory cache misses is to identify subsets of regularly used files and then hoard them to local disk. A background task will be responsible for monitoring external file changes and to keep the local copies reasonably fresh. External file changes can be detected by snooping the current table. Hoarding has been investigated in Coda [20] and later refined in SEER [22]. We plan to integrate ideas from their systems into JetFile.

Hoarding can only be expected to work well with regularly used files. There must also be a mechanism to avoid compulsory cache misses on files that are not subject to hoarding. We suggest that files are prefetched as a background task controlled by network parameters delay and available bandwidth. If the delay is long and bandwidth is plentiful it makes sense to prefetch files to decrease the number of compulsory cache misses.

11 Open Issues and Limitations

JetFile requires file managers to join a multicast group for each file they actively use or serve. This implies that routers will be forced to manage a large multicast routing state.

The number of multicast addresses used may be decreased by hashing FileIDs onto a smaller range. This has the disadvantage of wasting network bandwidth, and to force file managers to filter out unwanted traffic. How to strike the balance between address space usage and probability of collision we do not know.

The concept of *wakeup messages* can be generalized to wakeup servers for files. A message can be sent to a file's corresponding volume address, the message will make passive servers join the file address, in effect activating servers. This idea can be further generalized. A message can be sent to an *organization address* to make passive servers join the volume address. A disadvantage of this approach is that *wakeup messages* will introduce a new type of delay.

Lastly, when writing this, we do not yet have any experience with large scale Inter-domain multicast routing. If the routers and multicast routing protocols of tomorrow will be able to cope with the load generated by JetFile remains to be seen.

12 Conclusions

The JetFile distributed file system combines new concepts from the networking world such as IP multicast routing and Scalable Reliable Multicast (SRM), as well as proven concepts from the distributed systems world such as caching and callbacks, to provide a scalable distributed file system for operation across the Internet or large intranets. JetFile is designed for ubiquitous distributed file access. To hide the effects of round-trip delays and transmissions errors, JetFile takes an optimistic approach to concurrency control. This is a key factor in JetFile's ability to work well over long high-speed networks as well as over high delay/high loss wireless networks.

Dynamic replication is used to localize traffic and distribute load. Replicas are synchronized, located, and retrieved using multicast techniques. JetFile assigns server duties to clients to avoid the often expensive effects of writing data through the local cache to a server. In the common case when files are not shared, this is the optimal means to avoid unwanted network effects such as long delays, packet loss, and bandwidth limitations. Multicast routing and SRM are used to keep communication to a minimum. In this way, files are easily updated at their replication sites.

Callback renewal is aggregated on a per-volume basis and shared between clients to reduce server load. Moreover, JetFile *callbacks* are stateless and best-effort. This implies that servers need not keep track of which hosts are caching particular files. Thus client caching can be much more aggressive. It should be pointed out, that because of the best-effort nature of our *callback* scheme, clients may, under some circumstances, not be aware of cache invalidity for up to 30 seconds.

We have implemented parts of the JetFile design and experimentally verified its performance over a local area network. Our measurements indicate that, using a standard benchmark, JetFile performance is close to that of a local disk-based file system.

13 Acknowledgments

We would like to thank our shepherd Tom Anderson and the anonymous reviewers for their constructive comments which lead to significant improvements of this paper.

We also thank Mikael Degermark for reading early versions of this paper and making many insightful comments.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, R. Y. Wang, *Serverless Network File Systems*, In Proceedings of the 15th ACM Symposium on Operating Systems Principles, 1995.

[2] M. G. Baker, J. Hartman, M. D. Kupfer, K. W. Shirriff, J. Ousterhout, *Measurements of a Distributed File System*, In Proceedings of the 13th ACM Symposium on Operating Systems Principles, 1991.

[3] T. Ballardie, P. Francis, J. Crowcroft, *Core Based Trees (CBT)*, In Proceedings of the ACM SIGCOMM 1993.

[4] D. Banks, C. Calamvokis, C. Dalton, A. Edwards, J. Lumley, G. Watson. *AAL5 at a Gigabit for a Kilobuck*. Journal of High Speed Networks, 3(2), pages 127-145, 1994.

[5] T. Billhartz, J. Cain, E. Farrey-Goudreau, D. Fieg, S. Batsell, *Performance and Resource Cost Comparisons for the CBT and PIM Multicast Routing Protocols*, IEEE Journal on Selected Areas in Communications, 15(3), Apr. 1997.

[6] K. Birman, A. Schiper, P. Stephenson, *Lightweight Causal and Atomic Group Multicast*, ACM Transactions on Computer Systems, 9(3), Aug. 1991.

[7] D. D., Clark, D. L. Tennenhouse, *Architectural considerations for a new generation of protocols*, In Proceedings of the ACM SIGCOMM 1990.

[8] S. Deering, *Host Extensions for IP Multicasting*, RFC 1112, Internet Engineering Task Force, 1989.

[9] S. Deering, *Multicast Routing in a Datagram Internetwork*, PhD thesis, Stanford University, Dec. 1991.

[10] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, L. Wei, *The PIM Architecture for Wide-Area Multicast Routing*, IEEE/ACM Transactions on Networking, 4(2), Apr. 1996.

[11] S. Deering, C. Partridge, D. Waitzman, *Distance Vector Multicast Routing Protocol*, RFC 1075, Internet Engineering Task Force, 1988.

[12] W. Fenner, *Internet Group Management Protocol, Version 2*, RFC 2236, Internet Engineering Task Force, 1997.

[13] S. Floyd, V. Jacobson, , C. Liu, S. McCanne, L. Zhang, *A Reliable Multicast Framework for Lightweight Sessions and Application Level Framing*, IEEE/ACM Transactions on Networking, 5(6), Dec. 1997.

[14] C. G. Gray, D. R. Cheriton, *Leases: an efficient fault-tolerant mechanism for distributed file cache consistency*, In Proceedings of the 12th ACM Symposium on Operating System Principles, 1989.

[15] B. Grönvall, I. Marsh, S. Pink, *A Multicastbased Distributed File System for the Internet*, In Proceedings of the 8th ACM European SIGOPS Workshop, 1996.

[16] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, M. J. West, *Scale and Performance in a Distributed File System*, ACM Transactions on Computer Systems, 6(1), Feb. 1988.

[17] M. L. Kazar, *Synchronization and Caching Issues in the Andrew File System* In Proceedings of the USENIX Winter Technical Conference, 1988.

[18] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Buttos, S. Chutani, C. F. Everhart, W. A. Mason, S. Tu, E. R. Zayas, *DEcorum file system architectural overview* In Proceedings of the Summer USENIX Technical Conference, 1990.

[19] J. J. Kistler, *Disconnected Operation in a Distributed File System*, PhD thesis, Carnegie Mellon University, May. 1993.

[20] J. J. Kistler, M. Satyanarayanan, *Disconnected Operation in the Coda File System* ACM Transactions on Computer Systems, 10(1), Feb. 1992.

[21] S. R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*, In Proceedings of the USENIX Summer Technical Conference, 1986.

[22] G. H. Kuenning, G. J. Popek, *Automated Hoarding for Mobile Computers*, In Proceedings of the 16th ACM Symposium on Operating Systems Principles, 1997.

[23] P. Kumar, M. Satyanarayanan, *Flexible and Safe Resolution of File Conflicts*, In Proceedings of the USENIX Winter Technical Conference, 1995

[24] S. Kumar, P. Radoslavov, D. Thaler, C. Alaettinoglu, D. Estrin, M. Handley, *The MASC/BGMP Architecture for Inter-domain Multicast Routing*, In Proceedings of the ACM SIGCOMM 1998.

[25] E. K. Lee, C. A. Thekkath, *Petal: Distributed virtual disks*, In Proceedings of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, 1996.

[26] S. J. Mullender, A. S. Tanenbaum, *A Distributed File Service Based on Optimistic Concurrency Control*, In Proceedings of the 10th ACM Symposium on Operating Systems Principles, 1985.

[27] L. Mummert, M. Ebling, M. Satyanarayanan, *Exploiting Weak Connectivity for Mobile File Access*, In Proceedings of the 15th ACM Symposium on Operating Systems Principles, 1995.

[28] D. P. Reed, L. Svobodova, *SWALLOW: A Distributed Data Storage System for a Local Network*, Local Networks for Computer Communications, North-Holland, Amsterdam 1981.

[29] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, G. Popek, *Resolving File Conflicts in the Ficus File System*, In Proceedings of the USENIX Summer Technical Conference, 1994.

[30] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, *Design and Implementation of the Sun Network Filesystem*, In Proceedings of the USENIX Summer Technical Conference, 1985.

[31] B. Schneier, *Applied Cryptography, Second Edition*, John Wiley & Sons, Inc, 1996.

[32] B. Sidebotham, *VOLUMES - The Andrew File System Data Structuring Primitive*, In Proceedings of the European Unix User Group Conference, Manchester, 1986.

[33] M. Spasojevic, M. Satyanarayanan, *An Empirical Study of a Wide-Area Distributed File System*, ACM Transactions on Computer Systems, 14(2), May. 1986.

[34] D. C. Steere, J. J. Kistler, M. Satyanarayanan, *Efficient User-Level File Cache Management on the Sun Vnode Interface*, In Proceedings of the USENIX Summer Technical Conference, 1990.

[35] J. S. Steiner, C. Neuman, J. I. Schiller, *Kerberos: An Authentication Service for Open Network Systems*, In Proceedings of the USENIX Winter Technical Conference, 1988.

[36] C. A. Thekkath, T. Mann, E. K. Lee, *Frangipani: A Scalable Distributed File System*, In Proceedings of the 16th ACM Symposium on Operating Systems Principles, 1997.

[37] R. Wang, T. Anderson, *xFS: A Wide Area Mass Storage File System* In Proceedings of the Fourth Workshop on Workstation Operating Systems, 1993.

Integrating Content-Based Access Mechanisms with Hierarchical File Systems

Burra Gopal*
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052
burrag@microsoft.com

Udi Manber †
Dept. of Computer Science
University of Arizona
Tucson, AZ 85721
udi@cs.arizona.edu

ABSTRACT

We present a new file system that combines name-based and content-based access to files at the same time. Our design allows both methods to be used at any time, thus preserving the benefits of both. Users can create their own name spaces based on queries, on explicit path names, or on any combination interleaved arbitrarily. All regular file operations – such as adding, deleting, or moving files – are supported in the same way, and in addition, query consistency is maintained and adapted to what the user is manually doing. One can add, remove, or move results of queries, and in general handle them as if they were regular files. This creates interesting new consistency problems, for which we suggest and implement solutions. Remote file systems or remote query systems (e.g., web search) can be integrated by users into their own coherent name spaces in a clean way. We believe that our design can serve as the basis for the future information-rich file systems, allowing users better handle on their information.

1 Introduction

One of the most important challenges to current operating systems is to provide convenient access to vast amounts of information. By convenience, we mean not only the ability to quickly transfer information from one place to another, but the ability to find the right information and deal with it. This is arguably a new problem due to the scale of available information. File systems that were designed when typical users had few Magabytes and hundreds of files to contend with are getting inadequate when Gigabytes and hundreds of thousands of files are the norm.

The way we access file systems has not changed much in the last 30 years. Most file systems are based on a hierarchical arrangement with access by explicit path names or browsing (i.e., going down and up the tree). Hierarchical file systems have been successful because they provided everything we needed. They were extended to network file systems (trying to keep this transparent to the users), and widely distributed file systems. Numerous added features – such as quick search for file names, symbolic links or shortcuts, and automatic compression and backup, to name a few – make file system access even more convenient.

However, current file systems are hard pressed to deal with the vast amount of available information that is already upon us. Not in the physical sense – it is still relatively easy to store and access information. But being able to make effective use of that information is becoming harder and harder. For example, although a lot of information is obtained through searches, integrating this information into a file system is still done mostly by hand with little support. We present in this paper a new method of attacking this problem. We introduce a new paradigm,

*Work done while at the University of Arizona.

†Supported in part by DARPA contract N66001-96-C-8615. New-address: Yahoo! Inc., 3420 Central Expressway, Santa Clara, CA 95051.

actually a combination of old paradigms, and report on a successful implementation of a file system that follows that paradigm.

Our starting point is the *semantic file system* (SFS) paradigm introduced by Gifford et al [7]. Semantic file systems provide access by queries. They support the creation of *virtual directories*, each pointing to files that satisfy a query. Virtual sub-directories can be built using pointers from the parent, making a hierarchy based on query refinement. Semantic file systems allow users to organize their files by content and provide means to do that conveniently. This is sorely needed, because beyond a certain scale limit, people cannot remember locations by explicit path names. After so many years, it is still amusing to see even experienced UNIX system administrators spend time trying /usr/lib, or was it /usr/local/lib, maybe /opt/local/etc/lib, or /opt/unsupported/lib? There are, of course, many search tools available, but organizing large file systems is still too hard. The web, of course, has raised this problem to new heights.

So why haven't semantic file systems caught on? Clearly, as in any innovation, it takes a long time for people to change paradigms, especially if this directly involves everyday's tasks. It is essential to provide a smooth transition, which is currently not available. In addition, hierarchical file systems offer strong features that are not supported by semantic file systems. So the natural question is "can we combine the two paradigms?" Can we build a file system that will have the benefits of both hierarchical *and* semantic file systems, and allow users to choose among their features at any time?

We want to allow the use of the file system as a regular traditional hierarchical file system with no need to change anything. The added features of a content-based access (**CBA**) should be optional under the control of the user. They can cover the whole file system, any part of it, or none at all. They can be discarded and added at any time. Consequently, we base our design on a hierarchical file system and add content-based access, rather than extend a given content-based mechanism [12].

The main contribution of this paper is to show that combining name and content-based access is possible and that it can be implemented efficiently and reasonably cleanly. Our main goal is convenient and intuitive integration of information, without tying ourselves into any one special model. We maintain the full power of hierarchical file systems, allow users to automatically or manually modify and refine query results, preserve consistency of results even under manual changes, and provide integrated flexible access to remote file systems or query systems.

The paper is organized as follows. In section 2 we introduce our new file system, **HAC**, which stands for **Hierarchy And Content**. We discuss the major design problems, and suggest solutions and tradeoffs. Section 3 discusses how HAC connects to remote file systems and query systems through our notion of *semantic mount points*. Section 4 describes the implementation of HAC and gives performance measures, and section 5 discusses related work. A lot more work is needed to make such a system a mainstream general-purpose file system. We believe that this paper makes a significant step towards this goal.

2 The Design of the HAC File System

2.1 A Running Example

To describe the design of HAC we will use one running example. Suppose that the user is working on a project involving the use of *fingerprints* (as one of the authors had). Information about the project may be found in email with its participants, in notes, articles, source code files, etc. Typically each of these will be stored in a different place, possibly on a different computer (e.g, a laptop or a network file server). The user may also have relevant information from previous projects or from other sources which the user may not even remember. Furthermore, important information can be obtained through a search of remote facilities. HAC allows to combine all relevant material in one *semantic* directory; let's call it **fingerprint**. We'll see how to

build it, maintain it, and use it later on.

2.2 Queries, Query-Results and Semantic Directories

Current and suggested file systems that provide query support treat the "name space" associated with the query-based access to files as logically different from the name space associated with path name-based access. This makes it very difficult (if not impossible) to offer both forms of naming within the same system. For example, it is not possible to create new files within the *virtual directories* of SFS [7], and it is not possible to combine *views* of Nebula with directories in the "underlying" file system [5]. The *Multi-structured Naming* system [12] comes close. It allows users to specify certain relationships between queries (or "labels") so that users can organize queries and their results in a hierarchy. (Unlike SFS, if two queries in this system are related to each other in a hierarchy, their query-results do not necessarily have to be related in any way.) However, they still do not have the freedom to group files of their choice together within a label: they must also think of a query that matches the contents of exactly these files (and no others), and associate the query with this label.

Our approach to this problem is radically different: instead of starting with a query-based naming system and imposing a hierarchy or other relationships on queries, we start with a hierarchical naming system and extend it to support query (content) based naming. We show that this approach has many advantages: it gives users a lot of flexibility and power, and at the same time it makes the system easy and intuitive to use.

The first step is to map queries and their results onto file system abstractions. For obvious reasons, we decided to map queries into directories in the HAC file system. We call such directories *semantic directories*. When users create a new semantic directory, they specify both its path name and its query. HAC then creates a new directory, associates it with the query, and contacts the CBA mechanism to evaluate the query. In the new directory, HAC automatically creates new symbolic links to all files that satisfy the query. These symbolic links can co-exist with other information in the semantic directory, including other symbolic links or other regular files. The symbolic links can also point to files in other semantic directories in the file system, or even to remote file systems. HAC also provides a mechanism by which the user can easily extract the results of the query from these files. Semantic directories provide the abstraction and utility of virtual directories, but in HAC they are also *regular* hierarchical directories for all purposes. Users can add files to them, modify them, run applications from them, and so on.

HAC allows both ordinary *syntactic* directories to co-exist in the same file system. Directories (whether semantic or syntactic) can be accessed by specifying path names, and they can contain files, sub-directories, symbolic links, etc., as usual. Semantic directories contain additional information that helps HAC to maintain them and keep them consistent with whatever the user is doing. The consistency problem is a new non-trivial problem that we discuss next.

2.3 Scope of Queries and Scope Consistency

In HAC, every query – and its corresponding semantic directory – has a *scope* which is the set of files over which the query is evaluated. A query does not return symbolic links to files that are outside its scope even if those files match the query. The scope of a query depends on the parent of the corresponding semantic directory. If **..../parent/child** is a path such that both **parent** and **child** are semantic directories, then the scope of **child** is defined to be the existing set of symbolic links in **parent**. This set of symbolic links is also called the scope "provided" by **parent**. The scope provided by the root of a HAC file system is defined to be all the files in that file system. A change in the scope provided by **parent**, for example, will also change the scope of **child**. In this case, we say that **child** *depends* on **parent**. Note that all directories in the file system directly or indirectly depend on the root.

By these definitions, the scope provided by a newly created child semantic directory is always a *refinement* of the scope provided by its parent [5, 7]. When a user creates a new semantic subdirectory, HAC guarantees that the new set of symbolic links in that directory is always a subset of the set of the existing symbolic links in its parent. In other words, HAC treats the sets of symbolic links in different semantic directories as *separate* entities whose contents depend on how these directories are related to each other hierarchically. Hence, semantic directories allow users to organize both files and results of queries in a hierarchical fashion.

Semantic directories also allow users to tune the results of queries according to their personal tastes. HAC interprets the existing set of symbolic links in a semantic directory as its existing ("current") query result. Since each query-result is a separate entity, users can *modify* the result of any query by (i) deleting some irrelevant links returned by the query, (ii) creating new links to files that have related information, but were missed by the query, or (iii) adding regular files to that directory. In our fingerprint example, users may want to add a set of C programs implementing fingerprints, email messages from a certain user or about a certain topic, and/or image files to the **fingerprint** semantic directory, even though these files do not match **fingerprint**'s query. They may also decide that news stories about a certain crime should be removed from **fingerprint** even though they do match the query. They can do that by making the query more complex (e.g., "fingerprint AND NOT murder"), but often it is easier to remove a few files manually. Users can also build email semantic directories, allowing a message to be in more than one directory (e.g., by sender, receipient, topic, and/or a combination).

No query system is perfect, and currently most are not even close. HAC gives users more power to customize and adjust. It allows users to refine queries by using either the query language *or* the file system directly. Both methods are valid and being able to apply both at any time makes HAC very powerful and intuitive. But there's a major problem with this freedom. Since HAC allows users to edit the results of queries, it is now possible for them to create a hierarchy of semantic directories that makes sense to them intuitively, but still violates the scope restrictions discussed earlier. We call this the *Scope Consistency Problem.*

As far as we know, no existing file system has addressed the scope consistency problem. The Semantic File System [7] and Nebula [5] do not allow users to modify results of queries without modifying the query or the files in the file system. Prospero [9] allows users complete freedom to define and manipulate queries (or "filters") and their results, but does not talk about enforcing any kind of consistency when results of queries are arranged in a hierarchy (or a graph). Search systems like Harvest [4] and various WWW search engines are geared to bring search results to users, but not to organize results in any meaningful way.

Our approach to this problem in one of the main contributions of this paper. We now describe our solution in detail and show that it gives rise to a powerful new paradigm. To begin with, we classify symbolic links in a semantic directory in three ways (this distinction, for the most part, is hidden from users):

Permanent symbolic links: links that were explicitly added by the user to the directory.

Transient symbolic links: links that were obtained by evaluating a query.

Prohibited symbolic links: links (whether transient or permanent) that were once present in the directory but at some point were explicitly deleted from it by the user. HAC will ensure that these links will not be implicitly added later without a direct action by the user.

Given a semantic directory **sd**, which is not the root of a HAC file system, we define the *scope restriction* on the set of symbolic links in **sd** as the following invariant:

1. *The set of transient symbolic links in* **sd** *is*

always a subset of the scope provided by its parent **parent**, and

2. **sd** *should have transient symbolic links to all the files in the scope provided by* **parent** *that satisfy* **sd***'s query, except for links that are explicitly prohibited in* **sd**.

Changes to **sd**'s scope can lead to a breakup of these invariants, a situation we call *scope-inconsistency*. This can happen, for example, whenever

1. a user modifies the set of symbolic links in **sd**'s parent **parent**,
2. a user moves **sd** toa different part of the file system,
3. there is a change in the scope of **parent**, or
4. a user changes the query of **sd** after he/she creates it. (HAC allows users to access and modify the query associated with a semantic directory.)

A major part of HAC is an algorithm to maintain scope consistency, which is briefly described below. First, HAC uses the CBA mechanism to re-evaluate **sd**'s query on all the files in its current scope. Then, from this result, HAC discards the links that occur in **sd**'s set of permanent and prohibited symbolic links. The links that remain are the new transient symbolic links of **sd**. Note that HAC does not add a prohibited symbolic link to the above result even if that link points to a file that is in **sd**'s scope and matches its query. Similarly, HAC does not delete a permanent symbolic link from **sd** even if that link points to a file that is no longer in **sd**'s scope or does not match its query. Also note that HAC re-computes only the set of transient symbolic links of **sd** — HAC does not change the set of permanent or prohibited symbolic links associated with **sd** [1]. When the algorithm modifies the set of transient symbolic links in **sd**, it changes the scope provided by **sd**. Hence, the algorithm will also re-evaluate the queries of all the directories which directly or indirectly depend on **sd**. These are the descendents of **sd** and are present in the sub-tree rooted at **sd**. Any top-down traversal of this sub-tree (e.g., a breadth-first search) gives us the order in which we must re-evaluate the queries.

We decided to define the set of transient symbolic links in **sd** to be a refinement of the scope provided by its parent **parent**. We rejected the idea of defining this set to be, say, the union of the transient symbolic links in **sd** and the scope provided by all its children. (In this case, **sd** will depend on its children, not the other way round.) If we use this definition, users can never add a symbolic link **sl** to a child of **sd** such that **sl** does not automatically belong to the scope provided by **sd**. In other words, we cannot take care of the possibility that some information cannot be classified in a strict hierarchical fashion. This is unacceptable. We also rejected the idea of defining the set of transient symbolic links in **sd** to be the union of the transient symbolic links in **sd** and all its children, since in that case, changes to the set of transient links in a child semantic directory can effect the set of transient links in a parent. This is counter-intuitive since in hierarchical file systems, changes to the contents of a subdirectory do not effect the contents of its parent in any way.

To conclude: we allow users to edit and fine-tune the results of queries without modifying the query since we feel that the query of a semantic directory is not as important as the set of symbolic links in it. The query is just a quick first step to obtain more or less the information users are looking for. On the other hand, the set of symbolic links in a semantic directory may be the result of many (possibly time-consuming) browsing and editing steps. Hence, HAC does not modify this set unless it is explicitly asked to do so. Moreover, with this design, HAC is responsible only for the transient symbolic links in the file system, while users are responsible for all the permanent and prohibited symbolic links. HAC gives advice and help — users decide how to organize their file system.

[1] HAC has special API routines to directly modify the set of permanent and prohibited symbolic links in semantic directories. Sophisticated users can use these routines to control the behavior of the scope consistency algorithm.

2.4 Data Consistency

Users can create, remove, rename (move), or modify any data in the file system at will. There is therefore a possibility that the set of transient symbolic links in a semantic directory **sd** may not represent the current result of evaluating its query. This gives rise to a *data-inconsistency* problem. Data inconsistencies manifest themselves in the following ways: (i) A query result can contain an invalid link to a file that no longer exists, has been renamed, or has been modified so that it no longer satisfies the query, (ii) A query result may not contain a link to a new or modified file when it actually should. For example, new email that match the fingerprint query should be added to that semantic directory, and at the same time if a certain matching file has been moved to an area outside the scope of the query (e.g., it was deemed old and moved to archive), it should be removed from the semantic directory.

Though HAC removes *scope*-inconsistencies from the file system as soon as possible, HAC does not remove data-inconsistencies instantly. We could have adopted such a policy, similar to databases, but we believe that file systems typically do not require it, and the extra cost (determining when files have changed, re-indexing files automatically, etc.) will not warrant it. At present, HAC invokes the CBA mechanism to reindex the file system periodically (say, once a day or once an hour), determined by the user. At reindexing time, all scope and data consistencies are settled. HAC also allows users to initiate reindexing at any time, and for any part of the file system. So, for example, users can decide to update certain semantic directories as soon as new mail comes in, but not when an application modifies some files in the file system. In future, we plan to explore more sophisticated mechanisms to enforce data consistency in file systems.

2.5 Using Existing Results in New Queries

In addition to dependencies based on the hierarchy, HAC allow users to define arbitrary dependencies by adding directories names to queries. This gives users the power to combine query-language expressions (searching) with edited query results (browsing) in a very powerful way by specifying path names of existing directories (syntactic or semantic) as part of their queries. If the query of a semantic directory **new** contains the name of another semantic directory **old**, then we say that **new** *depends* on **old**, or **new** *refers* to **old**. Notice that dependencies are transitive. That is, if **old** depends on **ancient**, then **new** depends on **ancient**. When the CBA mechanism evaluates such a query, it can use HAC's API to determine which parts of the query contain path names of directories, and which parts contain search expressions. When it encounters the path name of a directory, it can use HAC's API to obtain the *existing* query-result (set of "pointers" to files) stored in that directory. Then, the CBA mechanism can operate on this set of pointers exactly as if it is the result of evaluating a search expression. In this way, users can easily augment the search capabilities of the CBA mechanism with their ability to customize information to their tastes.

One complication that arises here is that path-names can change when users rename directories. In the above example, if **old** is renamed as **old'**, **new**'s query is inconsistent since it refers to the old path name **old**. To solve this problem, HAC maintains a global mapping of unique identifiers to directory path-names, and stores only the identifers in actual queries. So, instead of updating the queries of all directories like **new** that depend on **old**, HAC simply updates the global map when **old** is renamed as **old'**.

When directory names are parts of queries, scope consistency becomes harder and trickier. For example, we must re-evaluate **new**'s query whenever the scope provided by **old** changes, even if **new** is **not** in the subtree rooted at **old**. We start with the definition of the *dependency graph* – it is the directed graph of dependencies between semantic directories. We do not allow

cycles to exist in this graph for obvious reasons. Updating the query-results cannot be done in an arbitrary order. We must use the order obtained from a *topological sort* of the dependency graph. There is always a valid topological sort since this is a *Directed Acyclic Graph* (DAG). The root of a HAC file system always occurs first in this order since all directories depend on it and the root does not depend on any other directory (the root does not have a query associated with it). Also note that there is no need for HAC to explicitly restrict the query result of a child directory to the scope provided by its parent. If users want this behavior, all they need to do is modify the query of the child directory to be: "<old query> **AND** <path-name of parent>" — HAC takes care of everything else. In fact, underneath the covers, this is exactly how we implement parent-child dependencies in the strict hierarchical scope consistency algorithm above. HAC gives users the freedom to chose whether they want strict hierarchical dependencies, DAG based dependencies, or both, interleaved arbitrarily.

Since we allow explicit scope consistency definitions along with implicit hierarchical scope consistency, one may argue that there is no need for the latter; we could leave it up to the user to specify the parent of a directory in its query. However, we feel that query-refinement based on path names is important for two reasons. First, tree-based classification of information is intuitive, and is sufficient for many real world scenarios. And second, most users may find tree-based classification simpler to understand because they do not have to worry about two structures — one based on path-names and one based on the dependency graph — at the same time.

3 Accessing Remote File and Query Systems

Another major benefit of HAC is its ability to cleanly access other HAC file systems, and other CBA mechanisms (possibly remote). In this section, we shall use *name space* to denote either a traditional file system (which provides path name-based access), a CBA mechanism, or a HAC file system. Connecting different file systems across a distributed system can be done with *mount points* [9, 10]. Mount points define new name spaces within which path names can be resolved. They allow different file systems to share certain directories so that they can access each other. HAC supports such mount points, which we call *syntactic mount points*. But we want to do more. We want to connect "semantically" so that we can evaluate queries against different name spaces, even if these name spaces *do not* allow us to organize information hierarchically (e.g., commercial search engines on the web). We want to allow users to use data from anywhere, create semantic directories anywhere, and in general treat the remote file systems and CBA mechanisms as if they were local. To achieve such rich, transparent connection, we must "decouple" the part of HAC that provides path name based access from the part that provides content based access, so that both can be used independently of each other. This is close to impossible to do if we restrict ourselves to syntactic mount points. We therefore introduced *semantic mount points* in HAC.

3.1 Semantic Mount Points

Let **Remote** be a remote file or query system. A semantic mount point **s.Remote** in a HAC file system **Local** connects queries within **Local** to results from **Remote**. Specifically, if the scope of a query within **Local** includes **s.Remote**, then it imports all the results asked within **Remote** with whatever query mechanism is used there. **s.Remote** provides an interface for *content-based* access to files in **Remote**. The power of a semantic mount point lies in the fact that the semantic directories created in it belong to the user's personal HAC file system, even though the symbolic links in these directories point to other (possibly remote) file systems. This allows users to create their own personal content-based classification of remote information. Furthermore, users can create physical files, semantic and syntactic directories, symbolic links, etc., as usual within semantic mount points. For instance, the physical files within

a semantic mount point are indexed by HAC, and they can match queries of semantic directories created outside the subtree rooted at the mount point. This level of integration of name and content based access gives users a tremendous amount of power – they can extract exactly what they want and organize it in exactly the way they like. Previous semantic file systems, such as SFS or Nebula, do not allow such rich integration of query results and physical files, semantic directories and mount points like HAC.

3.2 Multiple Semantic Mount Points

Just as it is possible to mount more than one file system on a syntactic mount point [10], it is also possible to mount more than one name space on a semantic mount point. HAC treats each such name space as an independent entity. The scope of queries asked within a *multiple* semantic mount point is simply a union of the scope provided by each mounted name space. Queries are evaluated independently in each name space and their results are treated as disjoint sets of symbolic links. The only restriction is that all name spaces mounted on a multiple semantic mount point must be accessible via the same query language. (Currently, HAC does not deal with overlapping name spaces or data, i.e., it does not resolve cases where two symbolic links might actually point to the same remote file, or to similar files.)

For example, suppose that we want to cover **Remote** and **Local** at the same time. All we need to do is create another semantic mount point **s.Local** as a multiple semantic mount point! There is no problem of cyclic reference here, because **s.Local** is just an interface to a CBA mechanism; it does not provide CBA on its own.

Syntactic and semantic mount points can be combined in various ways to share information by both name and content. Getting back to our fingerprint example, we may have access to a digital library with scientific articles. We can add a semantic mount point associated with a query for "fingerprint" (or a more complex query), thus ensuring that our knowledge of the subject is up to date (at least with the library). There will probably be other sources as well, and we may need to form different queries depending on the source. We may want to have syntactic mount points to all these sources and search there manually once in a while, but in addition HAC allows a user to build remote semantic directories for each source (or one for all of them), and have a better access to and better integration with this information. For example, one can "remove" certain results of no interest, add comments, add results from other places, etc. Other users (e.g., coworkers on the same project) can use syntactic mount points to browse through one user's personal classification (instead of doing the searches themselves) and retrieve relevant information. It is also possible to collect the names, queries and query-results of many semantic directories of many users in a central database that *itself* can be indexed and searched. Users can browse and search this database and find others who have similar tastes as they have. This may help them find what they are looking for even more quickly. Finally, users can add their favorite books, articles, memoirs, shortcuts to other information, etc., to their personal HAC file systems, index and search them, and export their file systems as mini-digital libraries to others. To conclude: semantic mount points give us a powerful new way to access the semantic aspect of information. They can be combined with syntactic mount points to yield a rich set of primitives for sharing information in a distributed system.

4 Implementation and Performance

We implemented HAC on top of a UNIX file system (SunOS) using *Glimpse* as the default CBA mechanism [8]. Our prototype contains about 25,000 lines of C code. It was implemented as a dynamically linked library (DLL) which can be accessed by all user-level applications. *No kernel modifications were used.* This made the design easier to experiment with, easier to port, and easier to convince people to use it. The obvious

disadvantage is a penalty in performance compared to a native UNIX file system, but as we will show, it is a small penalty. We start with a brief overview of the implementation.

HAC allows users to define their own personal name spaces (i.e., a personal file system). HAC uses this name space to resolve the users' path names and evaluate their queries. This name space exists within a directory in the UNIX file system. HAC intercepts all file system calls that access this directory or its contents, and provides a transparent interface to all applications. Well-known file system commands, such as `cd`, `ls`, `mkdir`, `mv`, `rm`, etc., can be used to access and manipulate objects in the file system in the usual way. HAC also provides additional commands that manipulate queries and semantic directories. These are for the most part intuitive extensions of regular file system commands. For example, `smkdir` creates a semantic directory, `smv` modifies the query of a directory and `sreadln` retrieves it, `scat` accepts a symbolic link in a semantic directory and returns the information in the corresponding file that matches the query of the directory, `smount` defines new syntactic and semantic mount points, and `ssync` re-evaluates the queries of all the directories that directly or indirectly depend on a given directory.

HAC interacts with UNIX using a well defined API which assumes very little about the native file system — HAC can be used even on "flat" file systems and file systems that do not support symbolic links. HAC interacts with Glimpse using another simple, well defined API. We believe that this API is general enough to integrate any CBA mechanism into HAC. Since HAC is a user-level file system, it does not contain any security and access-control features of its own: it borrows them from the underlying operating system.

We ran several experiments to determine the overhead to extended file system operations compared with regular file systems and/or regular glimpse queries. In the first experiment, we measured the overhead when we used HAC as a syntactic file system like UNIX and ran the Andrew Benchmark [11] on both systems. The Andrew Benchmark has been used as a standard to evaluate the performance of many new file systems. The benchmark has 5 phases: (i)**Makedir**: constructs a destination directory hierarchy that is identical to the source directory hierarchy, (ii)**Copy**: copies each file in the source hierarchy to the destination hierarchy, (iii)**Scan**: recursively traverses the whole destination hierarchy and examines the status of every file in the hierarchy without reading the actual data in the files, (iv)**Read**: reads every byte of every file in the destination hierarchy, and (v)**Make**: compiles and links the files in the destination hierarchy. The results for HAC are shown in tables 1 and 2 below:

From table 1, we see that phases 1 and 2 have the maximum overhead. This is because in phase 1, when HAC creates a new directory, it also creates and initializes (to "empty") the data structures that store its query, its query-result, and its set of permanent and prohibited symbolic links. HAC keeps track of the name of this directory in a global map so that it can track changes to the structure of the file system. Finally, HAC creates a new (empty) node for the directory in the dependency graph. (All of these are stored in the disk and require extra I/O operations.) In phase 2, when HAC creates a new file, it also initializes the open file-descriptor and the attribute-cache for that file. (This is stored in UNIX *shared memory* so that different processes can access it.) This helps to speed up Scan and Read operations on that file. Phases 3 and 4 have a medium overhead. In phase 3, HAC accesses the attribute-cache to retrieve the appropriate status information, and in phase 4, HAC accesses and updates the per-process file-descriptor table to implement the read-operation. Phase 5 has the least overhead since it is computationally intensive. On the whole, HAC is about 46 % slower than UNIX. From table 2 HAC is only slightly slower than the Jade [10] and Pseudo [13] file systems (both of which are user-level file systems like HAC). We also calculated the space overhead to store HAC's data structures (the extra information needed for each directory mentioned above, along with a fixed amount of book-keeping information). In the example we used, HAC required 222 KB while UNIX needs 210 KB. This is about 5 % more. The average amount of shared memory needed per process

File System	Makedir	Copy	Scan	Read	Make	Total
UNIX	2s	5s	5s	8s	19s	39s
HAC	4s	9s	8s	14s	22s	57s

Table 1: Results of Andrew Benchmark

File System	% Slowdown
Jade FS	36
Pseudo FS	33-41
HAC FS	46

Table 2: Comparison with other File Systems

(including the attribute cache and the descriptor table) is about 16KB. Both these overheads are negligible. We believe that HAC's performance is quite reasonable since unlike the other two file systems, HAC must also create and maintain data-structures that provide content-based access to files.

In the second experiment, we first used Glimpse to index a database consisting of over 17000 files that occupy about 150 MB. We ran the indexing mechanism directly over UNIX to get an estimate of the time Glimpse takes to index the database and the space needed to store the index. We then indexed a different copy of the same database by using the HAC file system library instead. The results are shown in table 3. We see that HAC has a 27 % time overhead and a 15 % space overhead: we believe that both of these are reasonable.

In the second part of this experiment, we used the `smkdir` command in HAC to create a semantic directory with a query **Q**. We also ran Glimpse through UNIX to search the above database for the same query. We chose three kinds of queries: (i) those that matched very few files, (ii) those that matched a lot of files, and (iii) those that matched an intermediate number of files. (We believe that queries of type (i) and (iii) are the most realistic — and the most useful.) The results are shown in Table 4.

For queries that matched very few files, Glimpse running on UNIX is more than 4 times as fast as HAC. This is because to interact with the CBA mechanism in HAC, we must create a semantic directory. We do not incur this overhead when we run Glimpse on UNIX to search files. While this may seem like a large overhead, in absolute terms it is very small. The overhead of creating a semantic directory reduces as the number of files that match the query increases. For queries that match an intermediate number of files, the overhead is about 15 %. For queries that match a lot of files, the overhead is only 2 %.

Regarding the space overhead, note that we need to store, with each semantic directory, the list of files matching the query. Instead of storing actual file names, which could add quite a bit of space, we use a compact representation of the list of all file names. This is part of HAC's API for the CBA mechanism. We currently use bitmaps since it is simple to implement and has speed advantages for Glimpse. The extra space we need per semantic directory is therefore N/8 Bytes, where N is the number of indexed files. This comes out to be about 2 KB in this experiment. We plan to improve this in future by using better sparse-set representations, so that it is possible to index a very large number of files.

5 Related Work

The first hierarchical file system to provide both name and content based access to files was the MIT Semantic File System (SFS) [7]. SFS introduced the concept of a *virtual directory*. The name of a virtual directory in SFS is a query, and the contents are symbolic links to files that

No. of files	17154
Size of files	149 Megabytes
Size of UNIX index	10 Megabytes
Size of HAC index	11.5 Megabytes
Time taken in UNIX	25 min
Time taken in HAC	31 min 48 sec

Table 3: Results of Indexing

No. of files that matched	1	6556	98
Time taken in UNIX	.45 sec	4 min 23 sec	7 sec
Time taken in HAC	2 sec	4 min 28 sec	8 sec

Table 4: Results of Searching

satisfy it. SFS assumes that queries are boolean **AND** combinations of "attribute-value" pairs, where an "attribute" is a typed field in the file system (e.g., "author:", "date:", etc.) and the "value" is a value this field can have (e.g., "John Doe", "3/12/97"). SFS always interprets the / path name separator between virtual directories as a conjunction operation. This feature can be used for query refinement.

SFS has many other novel features: (i) it caches the contents of different virtual directories to save query processing costs, (ii) it has special *transducer* programs that extract attributes and values from files in the file system to help with the indexing process (it also allows users to define their own transducers if necessary), and (iii) it has mechanisms to keep queries and their results consistent when there are changes to the files in the physical file system. However, SFS has some disadvantages. First, it assumes that queries are always conjunctions of attribute-value pairs, which makes it difficult to integrate arbitrary CBA mechanisms into the SFS. Second, virtual directories do not reside in the physical file system. Hence, users must use virtual directories to organize results of queries, but use real directories in the underlying file system to organize files. Third, SFS does not allow users to customize the results of queries according to their tastes without modifying queries or files in the file system. And finally, SFS does not provide a mechanism by which users can share their content-based classification of information with each other.

Other file systems follow in the footsteps of SFS. The Nebula File System [5] also assumes that files can be viewed as collections of attribute-value tuples. Queries in Nebula, however, can be arbitrary search expressions, not just boolean ANDs as in SFS. Nebula replaces the traditional idea of a fixed directory hierarchy by dynamic *views* of this hierarchy that can classify files in the underlying file system. A view is similar to a virtual directory: it has a query associated with it and contains pointers to files that satisfy the query. However, every view also has a "scope" which is defined to be a set of views. When Nebula evaluates the query of a view, it searches only those files which are referred to by the views in its scope. Nebula allows users to organize views in a DAG instead of a tree like SFS. Users can also alter the structure of this DAG by changing the scopes of views without changing their queries. This allows users to customize the contents of their views. Nebula has means to keep the contents of views consistent when there are changes to the data in the file system. It also allows users to share their views with each other. Though Nebula has many advantages, note that views are not a part of the underlying physical file system and cannot be used to organize data. Also note that Nebula does not allow users to group pointers to arbitrary files together and put them in a view: the files must satisfy the query associated with the view. Hence, users cannot modify results of queries to customize them according to their tastes.

Another example is the Multistructured naming system [12]. It tries to blend hierarchical or graph structured naming (e.g., the UNIX file system) with flat attribute or set based naming (e.g., SFS). It attempts to combine the "sense of *place*" present in graph-based naming with the ability of set-based naming to retrieve files using any combination of information about them. In this system, every query has a *label*, which is simply an alias for the query. Users can then impose "ancestor-descendent" (and other) relationships on labels, and selectively loosen these relationships, so that users can name files by specifying (i) either path names that contain labels, or (ii) a list of queries the files satisfy, or both, in arbitrary order. Multistructured naming allows users to access each others' personal name spaces and share information. Note, however, that it is not possible to group arbitrary files together and assign them a label. Like views in Nebula, a label must always have a query associated with it and can refer to only those files that satisfy this query. Hence, labels are not as powerful as directories in a regular file system. An important limitation of the above systems is that they do not provide a way to decouple name based access from content based access. This makes it difficult for a user to gather information from different CBA mechanisms (with possibly different query languages), and create a personal classification of this information using a *single* file system.

The Prospero file system [9] uses another approach: it allows each user to create his/her own personal graph-structured name space (called a *virtual file system*) that can refer to files in one or more existing graph-structured physical file systems. Users can also access the name spaces of other users. In both virtual and physical file systems, "nodes" are directories and contain files or pointers to other (virtual or physical) files, while "links" are used to connect nodes with each other. The novelty of Prospero is that users can associate *filters* with links in their virtual file systems. A Filter is an arbitrary program that can alter users' perception of the contents of the directory (node) the link points to (this is called the *target* directory of that link). The input of the filter is the target directory and the files and links it contains, while the output is a set of links that point to new directories whose contents are derived from the contents of the target directory. This output is called a *view* of the target directory. Note that since a filter is an arbitrary program, it can access not only its input, but other virtual and physical directories as well. Prospero also allows users to compose the filter associated with one link with the filter associated with another link, so that they can specify the view of the directory pointed to by the first link as a function of the view of the directory pointed to by the second link. Users can execute filters and derive views that classify information according to their personal tastes. Prospero's filters, therefore, are powerful tools for information retrieval. Their only drawback is that filters *must* be written and executed by the user. Prospero does not ensure that the views of target directories are up-to-date when there are changes to (i) the contents of these directories, (ii) the filters associated with links to these directories, or (iii) the filters of other links that are composed with the filters mentioned in (ii). That is, Prospero does not offer consistency guarantees of any kind — users must execute the appropriate filters at the appropriate time to ensure consistency.

The Synopsis File System ([2], [3]) provides a secure access mechanism to retrieve and manipulate large amounts of data within a wide-area file system. It hides the heterogeneity of data behind a logical interface to information based on typed *synopses*. Each synopsis is an *object* that encapsulates information about a single file in the form of *attributes* indexed for fast search and retrieval. The extensible type system allows users to define *methods* on each synopsis for customized display, access and manipulation of the synopsis content and the associated file. Since a synopsis is an object, its attributes and methods can also be inherited (composed) from other those of other synopses. A collection of synopses can be combined into a *digest*, that provides topic-based searches. Synopses and digests together can make content based access over very large heterogenous file systems more meaningful. Together, they can form the basis for locating and organizing information.

Like Nebula, the Synopsis File System introduces new abstractions to encapsulate informa-

tion based on content. And like Prospero, it allows users to define how they want to organize and manipulate this information. However, it does not define how a user's hierarchical organization of information is kept consistent when the structure of the hierarchy changes (e.g., what happens if you interchange child and parent synopses in the synopsis hierarchy). That is, consistency criteria are specific to each synopsis object – not to the Synopsis File System as a whole. HAC, on the other hand, defines and enforces a "global" consistency criteria based on the hierarchy, and fully integrates path-name and content-based access in a file system. Though their basic approaches are different, we believe that HAC and the Synopsis File System can be used in conjunction to yield a very powerful tool for information retrieval.

There are several other systems that address related issues [9, 4, 6]. In general, systems that are very flexible and powerful like Prospero do not have a consistency model, and systems that are intuitive and simple like the SFS offer consistency guarantees but are not as powerful and do not allow users to organize the information retrieved by name and content using the same file system. We believe that the HAC file system meets both these needs.

6 Conclusions

We have shown in this paper that it is possible, with reasonable overhead, to combine name-based and content-based access to files at the same time, while preserving the main benefits of both methods. We identified several scope and consistency problems, suggested solutions, and described an implementation. This is obviously not the last word on this topic. More work needs to be done to convince people to add a major paradigm to their daily arsenal of tools. A major missing piece is usability testing, which we have not performed to date. Only when a working version is widely distributed and used can we determine how beneficial is this approach.

The implementation described in the paper was geared towards personal file systems, and as is it is not scalable to very large file systems (e.g., Internet wide). This is true for many existing file systems. In particular, our decision not to modify anything at the kernel level (so the system will be easier to distribute) adds quite a bit to the overhead. Implementing our ideas in the context of a large scale Intranet, for example, will be a major next step. We believe it is possible and very desirable. The current situation of server-based search facilities and user-based file systems with almost no connection between them can be and should be improved.

References

[1] T. Berners-Lee, R. Calliau, and B. Pollermann. World-wide web: The information universe. *Electronic Networking: Research, Applications, and Policy*, 2:52–58, Spring 1992.

[2] M. Bowman and R. John, "The Synopsis File System: From Files to File Objects," In *Workshop on Distributed Object and Mobile Code*, Boston, MA, June, 1996.

[3] Mic Bowman, "Managing Diversity in Wide-Area File Systems". *Second IEEE Metadata Conference*, Silver Spring, Maryland. September 1997.

[4] M. Bowman, P. Danzig, D. Hardy, U. Manber, and M. Schwartz, The Harvest information discovery and access system, *Computer Networks and ISDN Systems*, **28** (1995), pp. 119-125.

[5] M. Bowman, Chanda Dharap, Mrinal Baruah, B. Camargo, and Sunil Potti, A file system for information management, In *Proc. Conf. on Intelligent Information Management Systems*, Washington, DC, June 1994.

[6] D. Cutting, D. Karger, and O. Pedersen. Constant Interaction-time Scatter/Gather Browsing of Very Large Document Collections. In *Proc. 16th Annual ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 126–134, 1993.

[7] D. Gifford, P. Jouvelot, M. Sheldon, and Jr J. O'Toole. Semantic file systems. In *Proc. 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA (October 1991), pp. 16-25.

[8] U. Manber and S. Wu. Glimpse: A tool to search through entire file systems. In *Usenix Winter 1994 Technical Conference*, San Francisco (January 1994), pp. 23-32.

[9] B. Neumann. The Prospero file system: A global file system based on the virtual system model. In *Proc. Usenix Workshop on File Systems*, May 1992.

[10] H. Rao and L. Peterson. Accessing files in an internet: The Jade file system. *IEEE Transactions on Software Engineering*, **19**(6):613–624, June 1993.

[11] M. Satyanarayanan Scalable, secure, and highly available distributed file access. *IEEE Computer*, **23**(5):9-21, May 1990.

[12] S. Sechrest and M. McClennen. Blending hierarchical and attribute-based file naming. In *Proc. 12th Intl. Conf. on Distributed Computer Systems*, Yokohama, Japan, June 1992.

[13] B. Welch and J. Ousterhout. Pseudo-File-Systems. Technical Report UCB/CSD 89/499, University of California, Berkeley, CA 1989.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, and C++, book and software reviews, summaries of sessions at USENIX conferences, and Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT – as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- Discount on BSDI, Inc. products.
- Discount on all publications and software from Prime Time Freeware.
- 20% discount on all titles from O'Reilly & Associates.
- Savings (10-20%) on selected titles from Academic Press, MIT Press, Morgan Kaufmann, O'Reilly & Associates, Prentice Hall, Sage Science Press , and John Wiley & Sons.
- Special subscription rate for *The Linux Journal* and *The Perl Journal.*
- The right to vote on matters affecting the Association, its bylaws,and election of its directors and officers.

Supporting Members of the USENIX Association:

Apunix Computer Services
Auspex Systems, Inc.
Cirrus Technologies
Cisco Systems Inc.
CyberSource Corporation
Deer Run Associates
Earthlink Network, Inc.
Hewlett-Packard India Software Operattions
Internet Security Systems, Inc.
Microsoft Research
NeoSoft, Inc
New Riders Press
Nimrod AS
O'Reilly & Associates
Performance Computing
Questra Consulting
Sendmail, Inc.
TeamQuest Corporation
UUNET Technologies, Inc.
Windows NT Systems Magazine
WITSEC, Inc.

Sage Supporting Members:

Atlantic Systems Group
Collective Technologies
Deer Run Associates
D. E. Shaw & Co.
Global Networking & Computing Inc.
Microsoft Research
New Riders Press
O'Reilly & Associates
Remedy Corporation
SysAdmin Magazine
Taos Mountain
TransQuest Technologies, Inc.

For further information about membership, conferences or publications, contact: USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA. Phone: 510-528-8649. Fax: 510-548-5738. Email: *office@usenix.org.*
URL: *http://www.usenix.org.*

ISBN 1-880446-39-1